Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Mechanized metatheory revisited

Dale Miller

Draft: September 30, 2018

Abstract When proof assistants and theorem provers implement the metatheory of logical systems, they must deal with a range of syntactic expressions (e.g., types, formulas, and proofs) that involve variable bindings. Since most mature proof assistants do not have built-in methods to treat bindings, they have been extended with various packages and libraries that allow them to encode such syntax using, for example, de Bruijn numerals. We put forward the argument that bindings are such an intimate aspect of the structure of expressions that they should be accounted for directly in the underlying programming language support for proof assistants and not via packages and libraries. We present an approach to designing programming languages and proof assistants that directly supports bindings in syntax. The roots of this approach can be found in the *mobility* of binders between term-level bindings, formula-level bindings (quantifiers), and proof-level bindings (eigenvariables). In particular, the combination of Church's approach to terms and formulas (found in his Simple Theory of Types) and Gentzen's approach to proofs (found in his sequent calculus) yields a framework for the interaction of bindings with a full range of logical connectives and quantifiers. We will also illustrate how that framework provides a direct and semantically clean treatment of computation and reasoning with syntax containing bindings. Some implemented systems, which support this intimate and built-in treatment of bindings, will be briefly described.

Keywords mechanized metatheory, λ-tree syntax, mobility of binders

1 Metatheory and its mechanization

Theorem proving—in both its interactive and automatic forms—has been applied in a wide range of domains. A frequent use of theorem provers is to formally establish correctness properties for specific programs: e.g., prove that a given program always terminates and correctly sorts a list or prove that a given loop satisfies a given invariant.

A more niche domain to which theorem proving is applied is that of the *metatheory* of programming languages. In this domain, one takes a formal definition of a particular

Inria Saclay and LIX/Ecole Polytechnique, Palaiseau, France

programming language's static semantics (e.g., typing), dynamic semantics (e.g., evaluation), and translation semantics (e.g., compilation) and establishes properties about all programs in that programming language. Typical examples of metatheorems are the following.

1. If evaluation attributes values U and V to program M, then U and V are equal (see, for example, [110, Theorem 2.4]). Thus, evaluation is a partial function.
2. If M is attributed the value V and it has the type A, then V has type A also. Thus, types are preserved when evaluating an expression (see, for example, [163]).
3. Applicative bisimulation for the programming language is a congruence (see, for example, [2,73]. Thus, equational-style rewriting can be used to reason about applicative bisimulation.

A theorem prover that is used for proving such metatheorems must deal with structures that are linguistic in nature: that is, metatheorems often need to quantify over programs, program phrases, types, values, terms, and formulas. A particularly challenging aspect of linguistic expressions, one which separates them from other inductive data structures (such as lists and binary trees), is their incorporation of bindings.

In fact, a number of research teams have used proof assistants to formally prove significant properties of entire programming languages. Such properties include type preservation, determinancy of evaluation, and the correctness of an OS microkernel and of various compilers: see, for example, [79,82,84,113].

The authors of the POPLmark challenge [12] had pointed out that proving metatheorems about programming languages is often a difficult task given the proof assistants available at that time (in 2005). In particular, their experiments with formulating the metatheory of programming languages within various proof assistants led them to urge the developers of proof assistants to improve their systems.

> Our conclusion from these experiments is that the relevant technology has developed *almost* to the point where it can be widely used by language researchers. We seek to push it over the threshold, making the use of proof tools common practice in programming language research—mechanized metatheory for the masses. [12]

These authors also acknowledge that poor support for binders in syntax was one problem that held back proof assistants from achieving even more widespread use by programming language researchers and practitioners.

In the decade following the POPLmark challenge, a number of approaches to representing syntax containing bindings have been proposed, analyzed, and applied to metatheory issues. These approaches go by names such as *locally nameless* [26], *nominal reasoning* [10,133,136,164], and *parametric higher-order abstract syntax* [29]. In the end, nothing canonical seems to have arisen: see [11,135] for detailed comparisons between different representational approaches. On the other hand, most of these approaches have been used to take existing mature proof assistants, such as Coq or Isabelle, and extend them with new packages, new techniques, new features, and/or new front-ends.

The incremental extension of mature proof assistants is only one way to address this issue. In this paper, we highlight another approach to mechanized metatheory and we use the following analogy to set the stage for that approach.

Early implementations of operating systems and distributed systems forced programmers to deal with concurrency, a feature not directly supported in early programming languages. Various treatments of concurrency and distributed computing

were addressed by adding to mature programming languages thread packages, remote procedure calls, and/or tuple spaces. Such additions made important contributions to what computer systems could do in concurrent settings. Nonetheless, early pioneers such as Dijkstra, Hoare, Milner, and Petri considered new ways to express and understand concurrency via formalisms such as CCS, CSP, Petri Nets, π-calculus, etc. These pioneers left the world of known and mature programming languages in an attempt to find natural and direct treatments of concurrent behavior. While the resulting process calculi did not provide a single, canonical approach to concurrency, their development and study have led to significant insight into the nature of computation and interaction.

In a similar spirit, we will examine here an approach to metatheory that is not based on extending mature theorem proving platforms. Instead, we look for means to compute and reason with bindings within syntax that arise directly from *logic* and *proof theory*, two topics that have a long tradition of allowing abstractions into the details of syntactic representations. There has been a number of technical papers and a few implementations that provide such an alternative approach to mechanized metatheory. The goal of this paper is not technical: instead, it is intended to provide an overview of this alternative approach.

2 Dropping mathematics as an intermediate

Before directly addressing some of the computational principles behind bindings in syntax, it seems prudent to describe and challenge the conventional design of a wide range of proof assistants.

Almost all ambitious theorem provers in use today follow the following two-step approach to reasoning about computation [96].

Step 1: *Implement mathematics.* This step is achieved by picking a general, well understood formal system. Common choices are first-order logic, set theory [118], higher-order logic [31,64], or some foundation for constructive mathematics, such as Martin-Löf type theory [34,35,86].

Step 2: *Reduce reasoning about computation to mathematics.* Computational systems can be encoded via a model-theoretic semantics (such as denotational semantics) or as an inductive definition over a proof system encoding, say, an operational semantics.

Placing (formalized) mathematics in the middle of this approach to reasoning about computational systems is problematic since traditional mathematical approaches assume *extensional* equality for sets and functions while computational settings may need to distinguish such objects based on *intensional* properties. The notion of *algorithm* is an example of this kind of distinction: there are many algorithms that can compute the same function (say, the function that sorts lists of integers). In a purely extensional treatment, functions are represented directly and descriptions of algorithms are secondary. If an intensional default can be managed instead, then function values are secondary (usually captured via the specification of evaluators or interpreters).

For an explicit example, consider whether or not the formula $\forall w.\ \lambda x.x \neq \lambda x.w$ is a theorem (assume that x and w are variables of some primitive type i). In a setting where λ-abstractions denote functions (which is the usual extensional treatment), this formula is equivalent to $\forall w.\neg\forall x.x = w$. As stated, we have not been provided enough information to answer this question: in particular, this formula is true if and only if the

domain type i is not a singleton. If, however, we are in a setting where λ-abstractions denote syntactic expressions, then it is sensible for this formula to be provable since no (capture-avoiding) substitution of an expression of type i for the w in $\lambda x.w$ can yield $\lambda x.x$. Taking this latter step means, of course, separating λ-abstraction from the mathematical notion of function.

A key methodological element of this paper is that we shall drop mathematics as an intermediate and attempt to find a direct and intimate connection between computation, reasoning, and logic.

Church's Simple Theory of Types [31] is one of the most significant and early steps taken in the design of a rich and expressive logic. In that paper, Church showed how it was possible to turn the tables on the usual presentation of terms and formulas in quantificational logic. Most presentations of quantification logic defined terms first and then formulas were defined to incorporate such terms (within atomic formulas). Church, however, defined the general notion of *simply typed λ-term* and defined formulas as a subset of such λ-terms, namely, those of type o. The resulting formal system provided an elegant way to reduce all formula-level bindings (e.g., the universal and existential quantifiers) to the λ-binder. His approach also immediately captured the binders used in the definite description operators and Hilbert's ϵ-operator. Church's presentation of formulas and terms are used in many active computational logic systems such as the HOL provers [66], Isabelle [119], and λProlog [99].

Actually, Church's 1940 paper introduced *two* higher-order logics. Both of these logics are based on the same notion of term and formulas and use the same inference rules—namely, $\beta\eta$-conversion, substitution, modus ponens, and $\forall$-generalization—but use different sets of axioms. The first of Church's logics is often called *elementary type theory* (ETT) [7] and involves using only axioms 1-6 which include the axioms for classical propositional logic as well as the basic rules for quantificational logic at higher-order (simple) types. The second of Church's logics is the aforementioned *simple theory of types* (STT). This logic arises by adding to ETT the axiom of choice, the existence of infinite sets, and the extensionality of functions (axioms 7-11 in [31]). Church's goal in strengthening ETT by adding these additional axioms was to position STT as a proper foundation for much of mathematics. Indeed, formal developments of significant parts of mathematics can be found in Andrews's textbook [8] and in systems such as HOL [64, 70].

When we speak of dropping mathematics as an intermediate, it is at this point that we wish to rewind the steps taken by Church (and implementers of some proof assistants): for the task of mechanized metatheory, we wish to return to ETT and not accept all of the mathematics oriented axioms.

3 Elementary type theory

ETT is an appealing starting place for its parsimony in addressing both bindings in formulas (quantification) and bindings in terms by mapping them both to bindings in the simply typed λ-calculus. It provides both support for (higher-order) quantification as well as for terms containing bindings. In addition, the equality theory of ETT is that of α, β, and η-conversion which means that both alphabetic changes of bound variable names and capture-avoiding substitutions are all accounted for by the logical rules underlying ETT. The proof theory for ETT has been well developed for both both intuitionistic and classical variants of ETT (Church's original version was based on classical

logic). Among the results know for ETT are cut-elimination [61,140,155], Herbrand's theorem and the soundness of Skolemization [91], completeness of resolution [6], and unification [75]. Subsets and variants of ETT have been implemented and employed in various computational logic systems. For example, the TPS theorem prover [105], the core of the Isabelle theorem prover [121], the logic programming language λProlog [99], and the proof system Minlog [147] are all based on various subsets of ETT. For more about the history of the automation of ETT and STT see the handbook article [19].

The simple types in ETT are best thought of as *syntactic categories* and that the arrow type $\gamma \to \gamma'$ is the syntactic category of abstractions of category γ over category γ'. Typing in this weak sense is essentially the same as Martin-Löf's notion of *arity types* [120]. In Church's logic, the type o (omicron) is the type of formulas: other primitive types provide for multisorted terms. For example, the universal quantifier $\forall_\gamma$ is not applied to a pair containing a variable of type γ and a formula (of type o) but rather to an abstraction of type $\gamma \to o$. Both $\forall_\gamma$ and $\exists_\gamma$ belong to the syntactic category $(\gamma \to o) \to o$. When using ETT to encode some object-level language, the terms and types of that language can be encoded as regular terms of two different primitive types denoting the syntactic categories of object-level term and object-level type.

Richer type systems, such as the dependently typed λ-calculi—known variously as LF, λP, and $\lambda\Pi$ [69,18]—are also important in a number of computational logic systems, such as Coq [21], Agda [25], and Twelf [128]. Although we shall limit the type system of our metalogic to be simple types, the intuitionistic variant of ETT is completely capable of faithfully encoding such dependently typed calculi [43,151–153].

To be useful as the foundation of a mechanized metatheory, ETT needs extensions. For example, ETT does not directly offer induction and coinduction which are both clearly important for any logic hoping to prove metatheoretic results. Using a proof-theoretic presentation of (the intuitionistic fragment of) ETT, Section 8 describes an extension to ETT in which term equality is treated as a *logical connective* (following the work by Schroeder-Heister [145] and Girard [62]) and inference rules for induction [89] and coinduction [14,156,160] are added. Section 9 presents a further extension to ETT with the addition of a *generic* quantifier [55,104,156].

In conclusion, we have explicitly ruled out Church's extension of ETT to STT as a proper foundation for specifying metatheory. Instead we shall illustrate that a separate extension to ETT—based on introducing inference rules for equality, fixed points, and ∇-quantification—satisfy many of the needs for an expressible and implementable logic for mechanizing metatheory. It is important to note that while STT is equipped to deal with the mathematical notion of function (given the use of the definite description choice operator and extensionality), the extensions to ETT we use here do not provide a rich notion of function. Instead, relations are used to directly encode computations and specifications. Of course, relations can encode functions: for example, the addition of two natural numbers can be encoded by the relation belonging to the syntactic category $nat \to nat \to nat \to o$ (assuming that nat is a primitive type for which there are the usual two constructors encoding zero and successor). In the weak setting of ETT, the syntactic category $nat \to nat \to nat$ does not contain the usual functional notion of addition. Fortunately, metatheory abounds with relations that may or may not be functional. For example, the following are all prominent relations in this setting: a program and its types, a process and its transitions, and a formula and its proofs.

4 How abstract is your syntax?

Two of the earliest formal treatments of the syntax of logical expressions were given by Gödel [63] and Church [31] and, in both of these cases, their formalization involved viewing formulas as strings of characters. Even in the 1970's, one could find logicians using strings as representations of formulas: for example, in [6], an atomic formula is defined as a formula-cum-string in which the leftmost non-bracket primitive symbol is a variable or parameter. Clearly, such a view of logical expressions contains too much information that is not semantically meaningful (e.g., white space, infix/prefix distinctions, brackets, parenthesis) and does not contain explicitly semantically relevant information (e.g., the function-argument relationship). For this reason, those working with syntactic expressions generally parse such expressions into *parse trees*: such trees discard much that is meaningless (e.g., the infix/prefix distinction) and record directly more meaningful information (e.g., the child relation denotes the function-argument relation). The *names* of bound variables are one form of "concrete nonsense" that generally remains in parse trees.

One way to get rid of bound variable names is to use de Bruijn's nameless dummy technique [36] in which (non-binding) occurrences of variables are replaced by positive integers that count the number of bindings above the variable occurrence through which one must move in order to find the correct binding site for that variable. While such an encoding makes the check for α-conversion easy, it can greatly complicate other operations, such as substitution, matching, and unification. While all such operations can be supported and implemented using the nameless dummy encoding [36, 83, 116], the bureaucracy needed to support that style of syntax clearly suggests that they are best address within the implementation of a framework and not in the framework itself.

We list four principles about syntax that will guide our further discussion.

> **Principle 1:** The names of bound variables should be treated in the same way we treat white space: they are artifacts of how we write expressions and they have no semantic content.

Of course, the name of variables are important for parsing and printing expressions (just as is white space) but such names should not be part of the meaning of an expression. This first principle simply repeats what we stated earlier. The second principle is a bit more concrete.

> **Principle 2:** All term-level and formula-level bindings are encoded using a single binder.

With this principle, we are adopting Church's approach [31] to binding in logic, namely, that one has only λ-abstraction and all other bindings are encoded using that binder. For example, the universally quantified expression $(\forall x.\, B\, x)$ is encoded as the expression $(\forall(\lambda x.\, B\, x))$, where $\forall$ is now treated as a constant of higher-type. Note that if B is an expression not containing x free, then this latter expression is η-equivalent to $(\forall\ B)$ and universal instantiation of that quantified expression with the term t is simply the result of using λ-normalization on the expression $(B\ t)$. In this way, many details about quantifiers can be reduced to details about λ-terms.

> **Principle 3:** There is no such thing as a free variable.

This principle is Alan Perlis's epigram 47 [124]. This principle acknowledges that every variable and constant is actually declared somewhere, and that that location serves as

its binding. This principle also suggests the following, which is the main novelty in this list of principles.

> **Principle 4:** Bindings have *mobility* and the equality theory of expressions must support such mobility [95, 99].

Since the first three principles are most likely familiar to the reader, we describe the last principle in more detail in the next section.

5 Mobility of bindings

We now illustrate the mobility of bindings by showing first how term-level binders can move to formula-level binders (quantifiers) and then move to proof-level binders (eigenvariables). We also illustrate how binders can move within term structures via simple rewriting.

5.1 Binder movement from terms to formulas to proofs

Gentzen's sequents are useful for describing the search for a proof since they explicitly maintain the "current set of assumptions and the current attempted consequence." For example, the sequent $\Delta \vdash B$ is the judgment that states that B is a consequence of the assumptions in Δ. A literal translation of Gentzen's sequents makes use of free variables. In particular, when attempting to prove a sequent with a universal quantifier on the right, the corresponding right introduction rule employs an *eigenvariable* that must be a "new" or "fresh" variable. For example, in the inference figure

$$\frac{B_1, \ldots, B_n \vdash B_0[v/x]}{B_1, \ldots, B_n \vdash \forall x_\gamma.B_0}\ \forall\mathcal{R},$$

the variable v is not free in the lower sequent but it may be free in the upper sequent. Gentzen called such new variables *eigenvariables*. Unfortunately, written this way, this inference figure violates the Perlis principle (Principle 3 in Section 4). Instead, we augment sequents with a prefix Σ that collects eigenvariables and *binds* them over the sequent. The universal-right introduction rule now reads as

$$\frac{\Sigma, v : \gamma\ :\ B_1, \ldots, B_n \vdash B_0[v/x]}{\Sigma\ :\ B_1, \ldots, B_n \vdash \forall x_\gamma.B_0}\ \forall\mathcal{R},$$

where we assume that the eigenvariable signature contains always distinct variables (as is always possible given α-conversion for binding constructs). As a result, sequents contain both assumptions and eigenvariables as well as the target goal to be proved. Eigenvariables are *sequent-level bindings*. (A second kind of sequent-level binding will be introduced in Section 9).

To illustrate the notion of binder mobility, consider specifying the typing relation that holds between untyped λ-terms and simple types. Since this problem deals with the two syntactic categories of expressions, we introduce two primitive types: tm is the type of terms encoding untyped λ-terms and ty is the type of terms encoding simple type expressions. Untyped λ-terms can be specified using two constructors

abs: $(tm \to tm) \to tm$ and app: $tm \to tm \to tm$ (note that there is no third constructor for treating variables). Untyped λ-terms are encoded as terms of type tm using the following translation function:

$$\lceil x \rceil = x, \qquad \lceil \lambda x.t \rceil = (abs\ (\lambda x.\lceil t \rceil)), \quad \text{and} \quad \lceil (t\ s) \rceil = (app\ \lceil t \rceil\ \lceil s \rceil).$$

The first clause here indicates that bound variables in untyped λ-terms are mapped to bound variables in the encoding. For example, the untyped λ-term $\lambda w.ww$ is encoded as $(abs\ \lambda w.\ app\ w\ w)$. This translation bijectively maps α-equivalence classes of untyped λ-terms to $\alpha\beta\eta$-equivalence classes of simply typed λ-terms of type tm: note that such adequacy results traditionally use $\beta\eta$-long normal forms of type terms to encode canonical term representations [69]. Scott's encoding [148] of untyped λ-terms using a certain domain D for which there were retracts between $[D \to D]$ and D is similar to our syntactic encoding here: namely, the syntactic category tm plays the role of D and the two constructors encode the two retracts $abs : (tm \to tm) \to tm$ and $app : tm \to (tm \to tm)$. Simple type expressions can be encoded by introducing two constants, say i: ty and $arrow$: $ty \to ty \to ty$. Let $of :\ tm \to ty \to o$ be the predicate encoding the typing relation between untyped terms and simple types. (Following Church [31], we use the type o as the type of formulas.)

The following inference rule is a plausible rule regarding typing.

$$\frac{\Sigma : \Delta, of\ t\ (arrow\ i\ i) \vdash C}{\Sigma : \Delta, \forall y(of\ t\ (arrow\ y\ y)) \vdash C}\ \forall L$$

This rule states (when reading it from premise to conclusion) that if the formula C follows from the assumption that t has type $(arrow\ i\ i)$ then C follows from the stronger assumption that t can be attributed the type $(arrow\ y\ y)$ for all instances of y. In this rule, the binding for y is instantiated: this inference rule is an example of Gentzen's rule for the introduction of the $\forall$ quantifier on the left.

The following formula can be used to specify what it means for a λ-abstraction to have an arrow type.

$$\forall B \forall y \forall y' [\forall x (of\ x\ y \supset of\ (Bx)\ y') \supset of\ (abs\ B)\ (arrow\ y\ y')]. \quad (*)$$

Now consider the following combination of inference rules.

$$\dfrac{\dfrac{\Sigma, x : \Delta, of\ \lceil x \rceil\ y \vdash of\ \lceil B \rceil\ y'}{\Sigma : \Delta \vdash \forall x (of\ \lceil x \rceil\ y \supset of\ \lceil B \rceil\ y')}\ \forall R,\ \supset R}{\Sigma : \Delta \vdash of\ \lceil \lambda x.B \rceil\ (y \to y')}\ \text{backchaining on } (*)$$

(Backchaining can be seen as a focused application of Gentzen-style inference rules acting on a formula in Δ [100]: we are assuming that the formula $(*)$ is a member of Δ.) These inferences illustrate how bindings can *move* during the construction of a proof. In this case, the term-level binding for x in the lower sequent can be seen as moving to the formula level binding for x in the middle sequent and then to the proof level binding (as an eigenvariable) for x in the upper sequent. Thus, a binding is not converted to a "free variable": it simply moves.

This mobility of bindings needs support from the equality theory of expressions. Clearly, equality already includes α-conversion by Property 1. As we shall now see, a

small amount of β-conversion is also needed. Rewriting these last inference rules using the definition of the $\lceil\cdot\rceil$ translation yields the following inference figures.

$$\dfrac{\dfrac{\Sigma, x : \Delta, of\ x\ y \vdash of\ (B\ x)\ y'}{\Sigma : \Delta \vdash \forall x(of\ x\ y \supset of\ (B\ x)\ y')}\ \forall R}{\Sigma : \Delta \vdash of\ (abs\ B)\ (arrow\ y\ y')}\ \text{backchaining}$$

Note that here B is a variable of arrow type $tm \to tm$ and that instances of these inference figures will create an instance of $(B\ x)$ that may be a β-redex: that β-redex has, however, a greatly restricted form. Also observe that the alternation of quantifiers implies that any instantiation of B leaves the β-redex $(B\ x)$ in the state where the argument x is not free in the instance of B: this is enforced by the fact that substitutions into formulas do not capture bound variables. Thus, the only form of β-conversion that is needed to support this notion of binding mobility is the so-called β_0-conversion [92], defined as $(\lambda y.t)x = t[x/y]$, provided that x is not free in $\lambda y.t$. (Note that this conversion is equivalent to $(\lambda x.t)x = t$ in the presence of α-conversion.)

Mobility of bindings is supported using β_0 since the internally bound variable y in the expression $(\lambda y.t)x$ is replaced by the externally bound variable x in the expression $t[x/y]$. Note that β_0 supports the following symmetric interpretation of λ-abstraction.

- If t is a term over the signature $\Sigma \cup \{x\}$ then λ-introduction yields the term $\lambda x.t$ which is a term over the signature Σ.
- If $\lambda x.s$ is a term over the signature Σ then the β_0-reduction of $((\lambda x.s)\ y)$ is a λ-elimination yielding $[x/y]t$, a term over the signature $\Sigma \cup \{y\}$.

Thus, β_0-reduction provides λ-abstraction with a rather weak form of functional interpretation: given a λ-abstraction and an increment to a signature, β_0 yields a term over the extended signature. The λ-abstraction has a dual interpretation since it takes a term over an incremented signature and hides that increment.

5.2 Binder movement within terms

To further illustrate how β_0 conversion supports the mobility of binders, consider how one specifies the following rewriting rule: given a universally quantified conjunction, rewrite it to be the conjunction of two universally quantified formulas. In this setting, we would write something like

$$(\forall(\lambda x.(A\ x \wedge B\ x))) \mapsto (\forall(\lambda x.A\ x)) \wedge (\forall(\lambda x.B\ x)),$$

where A and B are schema variables. To rewrite an expression such as $(\forall\lambda z(p\ z\ z \wedge q\ a\ z))$ (where p, q, and a are constants), we first need to use β_0-expansion to get the expression

$$(\forall\lambda z[((\lambda w.\ p\ w\ w)z) \wedge ((\lambda w.\ q\ a\ w)z)]).$$

At this point, the variables A and B in the rewriting rule can be instantiated by the terms $\lambda w.\ p\ w\ w$ and $\lambda w.\ q\ a\ w$, respectively, which yields the rewritten expression

$$(\forall(\lambda x.(\lambda w.\ p\ w\ w)\ x)) \wedge (\forall(\lambda x.(\lambda w.\ q\ a\ w)\ x)).$$

Finally, a β_0-reduction yields the expected expression $(\forall\lambda x.\ p\ x\ x) \wedge (\forall\lambda x.\ q\ a\ x)$. Note that at no time did a bound variable become unbound.

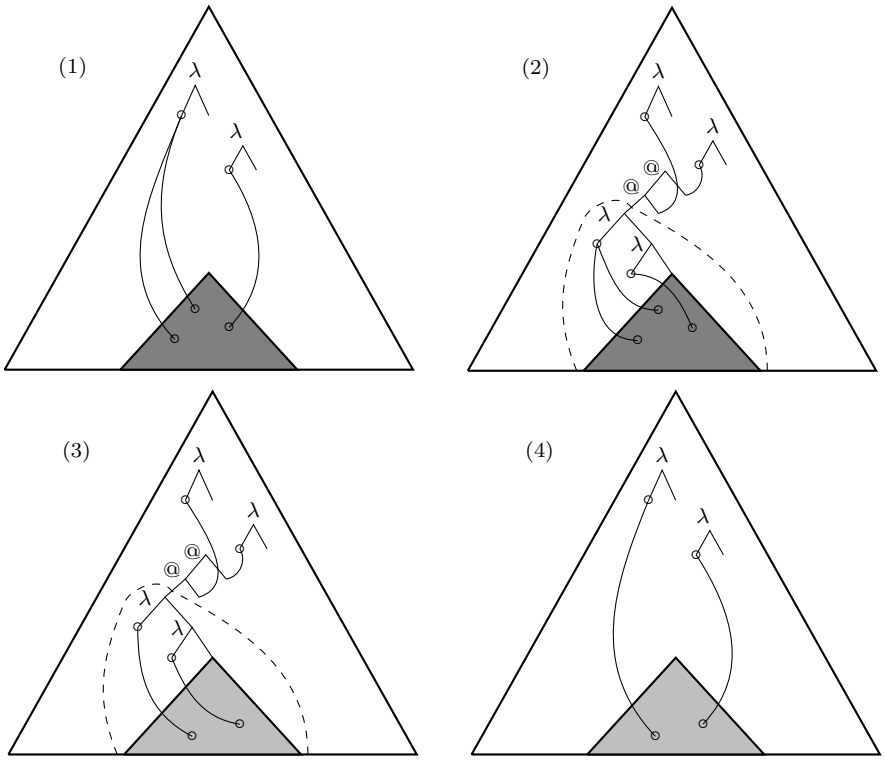

Fig. 1 Moving from (1) to (2) involves β_0-expansions; moving from (2) to (3) involves a replacing a λ-abstracted term; and moving from (3) to (4) involves β_0-reduction. Here, @ denotes application nodes.

Figure 1 graphically illustrates this process of rewriting in the presence of bindings. Assume that we have a closed term (illustrated in (1) of Figure 1 as a large triangle) and that we wish to replace an open subterm (the dark gray triangle) with another term (the light gray triangle in image (4)). Since the subterm in (1) contains occurrences of two bound variables, we write that subterm as $t(x, y)$ (where we assign the names x and y to those two bindings). When moving from image (1) to (2), we use β_0-expansion to replace $t(x, y)$ with $(\lambda u \lambda v.t(u, v))xy$. Note that the subterm $\lambda u \lambda v.t(u, v)$ is closed and, as a result, it can be rewritten to, say, $\lambda u \lambda v.s(u, v)$ (yielding (3)). Finally, β_0-reduction yields the term illustrated in (4). Thus, β_0-expansion and reduction allows a subterm be released from its dependency on bindings in its environment by changing those dependencies into local bound variables. Of course, instead of simply rewriting the open term t to the open term s, we needed to rewrite the closed abstraction $\lambda u \lambda v.t(u, v)$ to the closed abstraction $\lambda u \lambda v.s(u, v)$.

6 Proof search provides a framework

From a proof-theoretic perspective, formal reasoning can be seen as a process that builds a (sequent calculus) proof. The cut rule (the use of both modus ponens and

lemmas) is a dominate inference rule when reasoning is seen in this fashion [57]. The *proof search* approach to computation [100] can also be seen as building sequent calculus proofs that do not contain the cut rule. In general, cut-elimination is not part of these approaches to computation or reasoning. With the growing use of formal systems to encode aspects of mathematical reasoning, there are starting to appear some applications of cut-elimination within the reasoning process: consider, for example, *proof mining* where formal proofs can be manipulated to extract mathematically useful information [80]. In section 11, we shall provide a different set of examples where cut-elimination is used to formally reason about computations specified using the proof-search paradigm.

One of the appealing aspects of using proof search to describe computation and reasoning is that it is possible to give a rich account of binder mobility (as illustrated in Section 5). Thus, this paradigm allows for specifying recursive programming over data with bindings as well as inductive reasoning about such specifications. As such, proof search within ETT (even when restricted to not allow predicate quantification) can accommodate all four principles dealing with abstract syntax that were listed in Section 4.

We shall refer to computation-as-cut-free-proof-search as the *logic programming* paradigm, following the foundations developed in [100]. The use of this term is not intended to be narrowed to specific implementations of logic programming, such as Prolog or λProlog: for example, both of those languages make use of depth-first search even though such a search regime is often inappropriate for general logic programs.

The use of logic programming principles in proof assistants pushes against usual practice: since the first LCF prover [65], many proof assistants have had intimate ties to functional programming. Furthermore, many theorem provers view proofs constructively in the sense that computational content of proofs can be translated into executable functional programs [20].

Most of the remainder of this paper provides an argument and some evidence that the proof search paradigm is an appropriate and appealing setting for mechanizing metatheory. Our focus will be on the *specification* of mechanized metatheory tasks and not on their *implementation*: it is completely possible that logic programming principles are used in specifications while a functional programming language is used to implement that specification language (for example, current implementations of λProlog and Abella are written in OCaml [1, 40, 141]).

6.1 Expressions versus values

Keeping with the theme mentioned in Section 2 that types denote *syntactic categories*, the terms of logic should then denote expressions. If we are representing expressions without bindings, then expressions denote themselves, in the sense of free algebras: for example, the equality $3 = 1 + 2$ fails to hold because the equality is placed between two different expressions. While this is a standard expectation in the logic programming paradigm, the functional programming paradigm recognizes this equality as holding since, in that paradigm, expressions do not denote themselves but their *value*. That is, in the functional programming paradigm, if we wish to speak of expressions, we would need to introduce a datatype for abstract syntax (e.g., parse trees) and then one would have different expressions for "three" and for "one plus two."

The treatment of syntax with bindings within the functional programming paradigm is generally limited to two different approaches. First, binders in syntax can be mapped to function abstractions: thus, abstract syntax may contain functions. More about this approach appears in Section 7. Second, one can build datatypes to denote syntax trees using different representations of bindings, such as strings-as-variable-names or de Bruijn's nameless dummies [36]. The implementer of such a datatype would also need to encode notions such as α-equality, free/bound distinctions, and capture-avoiding substitution. Such an approach to encoding syntax with bindings is usually challenged when attempting to treat Principles 3 and 4 of Section 4. In order to support the notion that there are no free variables, contexts must be introduced and used as devices for encoding bindings: such bindings usually become additional data-structures and additional arguments and technical devices that must be treated with care. With its formal treatment of contexts (based on Contextual Model Type Theory [117]), the Beluga programming language [129] represents the state-of-the-art in this approach to syntax.

The logic programming paradigm with its emphasis on expressions instead of values provides another approach to treating syntax containing bindings that simply involves adopting an equality theory on expressions. In particular, by supporting both α-conversion and β_0-conversion it is possible for both Principle 1 and 4 to be supported. It has been know since the late 1980's that the logic programming paradigm can support the theory of α, η, and full β–conversions and, as such, it can support a suitably abstract approach to syntax with bindings: for example, the systems λProlog [99,114], Twelf [128], and Isabelle [122] provide such a proof search based approach to abstract syntax. While unification of simply typed λ-terms modulo $\alpha\beta\eta$ is undecidable in general [74], the systematic search for unifiers has been described [75]. It is also known that within the *higher-order pattern unification* restriction, unification modulo $\alpha\beta_0\eta$ is not only decidable and unary but it is also complete for unification modulo $\alpha\beta\eta$ [92]. This restricted form of unification is all that is needed to automatically support the kind of term processing illustrated in Figure 1.

6.2 Dominance of relational specifications

The focus of most efforts to mechanize metatheory is to build tools that make it possible to prove various properties of entire programming languages or specification languages (such as the λ-calculus and the π-calculus). The static semantics of such languages is usually presented as typing systems. Their dynamic semantics is usually presented as either *small step* semantics, such as is used in structural operational semantics (SOS) [134], or as *big step* semantics, such as is used in natural semantic [77]. In all of these styles of semantic specifications, relations and not functions are the direct target of specifications. For example, the specification of proof systems and type systems use binary predicates such as $\Xi \vdash B$ or $T\colon \gamma$. A binary relation, such as $M \Downarrow V$, is also used when specifying the evaluation of, say, a functional program M to a value V. In case it holds that evaluation is a (partial) function, then it is a metatheorem that

$$\forall M\, \forall V\, \forall V'\ [M \Downarrow V \wedge M \Downarrow V' \supset V = V']$$

Relations and not functions are the usual specification vehicle for capturing a range of programming semantics.

$$\frac{P \xrightarrow{A} P'}{P + Q \xrightarrow{A} P'} \qquad \frac{Q \xrightarrow{A} Q'}{P + Q \xrightarrow{A} Q'} \qquad \frac{P \xrightarrow{A} P'}{P \mid Q \xrightarrow{A} P' \mid Q} \qquad \frac{Q \xrightarrow{A} Q'}{P \mid Q \xrightarrow{A} P \mid Q'}$$

$$\frac{P \xrightarrow{A} P' \qquad Q \xrightarrow{\bar{A}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

Fig. 2 Some of the rules defining the labeled transitions for CCS. Tokens starting with a capital letter are schematic variables.

```
kind proc, act      type.

type tau            act.
type bar            act -> act.

type plus, par      proc -> proc -> proc.
type one            proc -> act  -> proc -> o.

one (plus P Q) A    P'           :- one P A P'.
one (plus P Q) A    Q'           :- one Q A Q'.
one (par  P Q) A    (par P' Q)   :- one P A P'.
one (par  P Q) A    (par P Q')   :- one Q A Q'.
one (par  P Q) tau  (par P' Q')  :- one P A P', one Q (bar A) Q'.
```

Fig. 3 The logic programming specification of SOS rules for CCS, written using the syntax of λProlog [99]. Here, the `kind` keyword declares `proc` and `act` as two syntactic categories denoting processes and actions, respectively. Tokens starting with a capital letter are variables that are universally quantified around the individual clauses.

For a concrete example, consider the small step semantic specification of CCS [106] which is usually given by defining the ternary relation of labeled transition systems $P \xrightarrow{a} Q$ between two processes P and Q and an action a. Figure 2 contains an SOS specification of the labeled transitions for CCS using inference rules. The connection between those inference rules and logic programming clauses is transparent: in particular, those inference rules can be written naturally as the logic programming clauses in Figure 3. The close connection between such semantic specifications and logic programming allows for the immediate animation of such specifications using common logic programming interpreters. For example, both typing judgments and SOS specifications have been animated via a Prolog interpreter in the Centaur project [33] and via a λProlog interpreter for computational systems employing binders [9,67,99].

The connection between semantic specifications and logic programs goes further than mere animation. Such logic programs can be taken as formal specifications which can then be used to prove properties about the computational systems they specify. For example, logic programs have been systematically transformed in meaning preserving ways in order to prove that a certain abstract machine implements a certain functional programming language [68]. Logic programs can also be used to specify and animate sophisticated transformations of functional programs such as closure conversion, code hoisting, and CPS transformations [172]. The Twelf system provided automated tools for reasoning about certain logic programs, thereby allowing direct proofs of, for example, progress theorems and type preservation [127,146]. A systematic approach to reasoning about logic programming specifications in Abella is described in Section 11.

6.3 Trading side conditions for more expressive logics

The inference rules used to specify both static semantics (e.g., typing) and dynamic semantics (e.g., small-step and big-step operational semantics) often contain an assortment of side conditions. Such side conditions can break and obscure the declarative nature of specifications: their presence can signal that a more expressive logical framework for specifications could be used.

The inference rules in Figure 2 for describing the transition system for CCS have no side conditions and their mapping into first-order Horn clauses (Figure 3) is unproblematic. Consider, however, some simple specifications regarding the untyped λ-calculus. The specification of call-by-value evaluation for untyped λ-terms can be written as

$$\frac{M \Downarrow \lambda x.R \qquad N \Downarrow U \qquad S \Downarrow V}{(M\ N) \Downarrow V} \quad \text{provided } S = R[U/x].$$

There, the side condition requires that S is the result of substituting U for the free variable x in R. Similarly, when specifying a typing discipline on untyped λ-terms, we typically see specifications such as

$$\frac{\Gamma, x\colon \gamma \vdash t\colon \sigma}{\Gamma \vdash \lambda x.t\colon \gamma \to \sigma} \quad \text{provided } x \notin fv(\Gamma).$$

Here, the side condition specifies that the variable x is not free the context Γ. In systems such as the π-calculus, which includes sophisticated uses of bindings, transition system come with numerous side conditions. Take, for example, the open inference rule [108]:

$$\frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}} \quad \text{provided } y \neq x,\ w \notin fv((y)P').$$

Here the side condition has two conditions on variable names appearing in that inference rule.

We will illustrate in Sections 8 and 10.2 how the side conditions in the inference rules above can all be eliminated simply by encoding those rules in a logic richer than first-order Horn clauses. In particular, the logic underlying λProlog, the *hereditary Harrop formulas* [100], provide an immediate specification of these rules, in part, because the intuitionistic logic theory of hereditary Harrop formulas directly supports binder mobility and rich quantified expressions.

6.4 Substitution lemmas for free

One of the reasons to use a logic to formalize static and dynamic semantic specifications is that that formalism can have significant formal properties of its own. For example, proof search as a computation paradigm usually constructs *cut-free proofs*. A famous metatheorem of intuitionistic logic is the *cut-elimination* theorem of Gentzen [57]: if properly used, the cut-elimination theorem can be seen as the "mother of all substitution lemmas." An example of a *substitution lemma* is the following: if $\lambda x.B$ has type $\gamma \to \gamma'$ and N has type γ then the result of substituting N for x in B, i.e., $[N/x]B$, has type γ'. To illustrate this claim, we return to the specification of the *of* predicate given in Section 5. This binary relation relates the syntactic categories *tm* (for untyped

λ-terms) and, say, *ty* (for simple type expressions). The logical specification of the *of* predicate might attribute integer type or list type to different expressions: for example, the following clause specifies typing for the non-empty list constructor ::.

$$\forall T\colon tm\ \forall L\colon tm\ \forall y\colon ty\ [of\ T\ y \supset of\ L\ (list\ y) \supset of\ (T :: L)\ (list\ y)].$$

Consider an attempt to prove the sequent $\Sigma : \Delta \vdash of\ (abs\ R)\ (t \to t')$ where the assumptions (the theory) contains only one rule for typing an abstraction and that this assumption is the clause $(*)$ from Section 5. Since the introduction rules for $\forall$ and $\supset$ are invertible, the sequent above is provable if and only if the sequent

$$\Sigma, x : \Delta, of\ x\ t \vdash of\ (R\ x)\ t'$$

is provable. Given that we are committed to using a proper logic (such as intuitionistic logic), it is the case that instantiating an eigenvariable in a provable sequent yields a provable sequent. Thus, the sequent $\Sigma : \Delta, of\ N\ t \vdash of\ (R\ N)\ t'$ must be provable. Thus, we have just shown, using nothing more than rather simple properties of logic that if

$$\Sigma : \Delta \vdash of\ (abs\ B)\ (t \to t') \quad \text{and} \quad \Sigma : \Delta \vdash of\ N\ t$$

then (using modus ponens) $\Sigma : \Delta \vdash of\ (B\ N)\ t'$. (Of course, instances of the term $(B\ N)$ can be β-redexes and the reduction of such redexes results in the substitution of N into the bound variable of the term that instantiates B.) Such lemmas about substitutions are common and often difficult to prove [5,168]: in this setting, this lemma is essentially an immediate consequent of using logic and logic programming principles. In Section 11, we illustrate how the two-level logic approach [56,90] implemented in the Abella theorem prover provides a general methodology that explicitly uses the cut-elimination theorem in this fashion.

7 λ-tree syntax

The term *higher-order abstract syntax* (HOAS) was originally defined as an approach to syntax that used "a simply typed λ-calculus enriched with products and polymorphism" [126].[1] It seems that few researchers currently use this term in a setting that includes products and polymorphism (although simple and dependently typed λ-calculus are often used). A subsequent paper identified HOAS as a technique "whereby variables of an object language are mapped to variables in the metalanguage" [128]. While this definition of HOAS seems the dominating one in the literature, this term is problematic for at least two reasons.

First, the adjective "higher-order" is both ambiguous (see [99, Section 1.3]) and unnecessary. In particular, the underlying notions of binder mobility and unification of terms discussed in Section 5 is valid without reference to typing and it is the order of a type that usually determines whether or not a variable is first-order or higher-order. When it comes to unification, in particular, it seems more appropriate to view pattern unification as a mild extension to first-order unification (which can be described without reference to types [92, Section 9.3]) than it is to view it as an extreme restriction to

[1] The reader who is not familiar with the term HOAS can safely skip to the last paragraph of this section.

"higher-order unification" (which, in general, requires knowing the types of variables). Thus, if types are not essential, why retain the adjective "higher-order"?

Second, this definition of HOAS is fundamentally ambiguous since the metalanguage (often the programming language) can vary a great deal. For example, if the metalanguage is a functional programming language or an intuitionistic type theory, the binding in syntax is usually mapped to the binding available for defining functions. In this setting, HOAS representation of syntax incorporates *function spaces* in expressions [37,71]. If the metalanguage is a logic programming language such as λProlog or Twelf, then the λ-abstraction available in those languages does not correspond to function spaces but to the weaker notion of hiding variables within a term, thereby producing a term of an abstracted syntactic type (see Section 2).

Referring to these different approaches to encoding syntax with the same expression leads to misleading statements in the literature, such as the following.

- The authors of [49] say that HOAS's "big drawback, in its original form at least, is that one looses the ability to define functions on syntax by structural recursion and to prove properties by structural induction—absolutely essential tools for our intended applications to operational semantics."
- In [142, p. 365], we find the statement that "higher-order abstract syntax used in a shallow embedding" when applied to "the π-calculus have been studied in Coq and λProlog. Unfortunately, higher-order datatypes are not recursive in a strict sense, due to the function in the continuations of binders. As a consequence, plain structural induction does not work, making syntax-analysis impossible. Even worse, in logical frameworks with object-level constructors, so-called *exotic* terms can be derived." Similar problems claimed about HOAS can also be found in [72,85].

If not read carefully, these negative conclusions about HOAS can be interpreted as applying to all methods of encoding object-level bindings into a metalevel binding. Since most higher-order languages allow functions to be defined with conditionals and recursion, syntax encoded with functions have "exotic" items (combinators related to conditionals and recursive definitions) injected into that syntax. While exotic terms can appear in Coq encodings [37], they are not possible in λProlog since it contains neither function spaces nor combinators for building functions using conditionals and recursion.

The term "λ-tree syntax" was introduced in [102] to avoid this ambiguous term. With its obvious parallel to the term "parse tree syntax," this term names an approach to the syntactic representation described in the previous sections that relies on the notions of *syntactic-categories-as-types*, $\alpha\beta_0\eta$-conversion, and mobility of bindings. In particular, λ-tree syntax is the form of HOAS available in the logic programming languages λProlog and Twelf. In both of those languages, it has long been known how to write relational specifications that compute by recursion over the syntax of expressions containing bindings. At the end of the 1990's, explicit reasoning about such relational specifications was part of the Twelf project [128] and was being developed for λProlog specifications following the "two-level logic approach" [88,90]. The Abella proof assistant was designed, in part, to reasoning about relational specifications: that prover is routinely used to prove inductive and coinductive theorems involving λ-tree syntax (see [15,51,56]). Abella is described in more detail in Section 11. Furthermore, various model-theoretic approaches to HOAS and λ-tree syntax are available: for example, Kripke-style models are used in [93,111] while a category of (covariant) presheaves is used in [48]. Finally, a couple of functional programming languages, namely, Bel-

uga [129] and MLTS [60], have been designed that introduce a binding construct that directly supports λ-tree syntax.

8 Computing and reasoning with λ-tree syntax

The proof theory of the intuitionistic version of ETT with higher-order (but not predicate) quantification provides a rich computational setting for the direct manipulation of λ-tree syntax. This section illustrates this claim.

8.1 Relational specifications using λ-tree syntax

A common method to specify the call-by-value evaluation of untyped λ-terms is using *natural semantics* [77] (also known as big-step semantics). For example, the following two inference rules (of which the second was already mentioned in Section 6.3)

$$\frac{}{\lambda x.R \Downarrow \lambda x.R} \qquad \frac{M \Downarrow \lambda x.R \quad N \Downarrow U \quad S \Downarrow V}{(M\ N) \Downarrow V} \text{ provided } S = R[N/x]$$

are commonly taken to be the definition of the binary relation $\cdot \Downarrow \cdot$ which relates two untyped λ-terms exactly when the second is the value (following the call-by-value strategy for reduction) of the first. Note that the rule for evaluating an application involves a side condition that refers to a (capture-avoiding) substitution.

Using the encoding in Section 5.1 of the untyped λ-calculus, the inference rules displayed above can be written as

$$\frac{}{abs\ R \Downarrow abs\ R} \qquad \frac{M \Downarrow (abs\ R) \quad N \Downarrow U \quad (R\ U) \Downarrow V}{(app\ M\ N) \Downarrow V}$$

In these inference rules, the schematic variables M, N, U, and V have type tm while the schematic variable R has type $tm \to tm$. Note that the side condition involving substitution, written as $S = R[N/x]$ above, has been removed and the application $(R\ N)$ appears instead. Since the equality theory of ETT contains β-conversion, instances of the term $(R\ U)$ are β-redexes if the instance of R is an abstraction. In that case, the result of performing a β-reduction would result in the formal substitution of the argument U into the abstracted variable of R, thereby correctly implementing the substitution proviso of the first pair of displayed inference rules above. These two inference rules are encoded naturally in intuitionistic logic as the following Horn clauses.

$$\forall R\,(eval\ (abs\ R)\ (abs\ R))$$
$$\forall M \forall N \forall U \forall V \forall R\,(eval\ M\ (abs\ R) \wedge eval\ N\ U \wedge eval\ (R\ U)\ V \supset eval\ (app\ M\ N)\ V)$$

Here, the infix notation $\cdot \Downarrow \cdot$ is replaced by the prefixed symbol *eval* and the type of the quantified variables M, N, U, V, and R is the same as when they were used as schematic variables above. The λProlog syntax for this specification is given in Figure 4: here kinds and types are explicitly declared and several standard logic programming conventions are used to displayed Horn clauses (upper case letters denote variables that are universally quantified around the full clause, conjunctions are written as a comma, and implication is written as `:-` denoting "implied-by").

```
kind  tm      type.
type  abs     (tm -> tm) -> tm.
type  app     tm -> tm -> tm.
type  eval    tm -> tm -> o.

eval (abs R) (abs R).
eval (app M N) V :- eval M (abs R), eval N U, eval (R U) V.
```

Fig. 4 The kind and type declarations and the logic program clauses specifying call-by-value evaluation for the untyped λ-calculus.

```
type copy                  tm -> tm -> o.
type subst (tm -> tm) -> tm -> tm -> o.

copy (app M N) (app P Q) :- copy M P, copy N Q.
copy (abs M)   (abs N)   :- pi x\ copy x x => copy (M x) (N x).

subst M T S :- pi x\ copy x T => copy (M x) S.
```

Fig. 5 A relational specification of object-level substitution.

8.2 A specification of object-level substitution and its correctness proof

As was illustrated above, the presence of β-conversion in ETT (and λProlog) makes it immediate to encode object-level substitution. Such substitution can be specified without reference to full β-conversion using only relational specifications and binder mobility. In particular, Figure 5 contains the specification of two predicates that can be used to capture such substitution. That figure makes use of three additional λProlog conventions: the backslash denotes λ-abstraction; `pi` denotes the logical constant for encoding universal quantification; and `=>` denotes implication. Following Church [31], the expression `pi x\` denotes the universal quantification of the variable `x`.

In isolation, the `copy` predicate encodes equality in the following sense. Let $\mathcal{C}$ denote the set of clauses in Figure 5. The judgment $\mathcal{C} \vdash$ `copy` M N is provable if and only if M and N are equal (that is, $\beta\eta$-convertible). The forward direction of this theorem can be proved by a simple induction on the uniform proof [100] of the judgment $\mathcal{C} \vdash$ `copy` M N. The converse is proved by induction on the structure of the $\beta\eta$-long normal form of terms of type tm. If the `copy` predicate is used hypothetically, as in the specification of the `subst` relation, then it can be used to specify substitution. The following is an immediate (and informal) proof of the following correctness statement for `subst`: $\mathcal{C} \vdash$ `subst` R M N is provable if and only if N is equal to the $\beta\eta$-long normal form of $(R\ M)$. The proof of the converse direction is, again, done by induction on the $\beta\eta$-long form of M (of type $tm \to tm$). The forward direction has an even more direct proof: the only way one can prove $\mathcal{C} \vdash$ `subst` R M N is to prove $\mathcal{C}, $`copy` x $M \vdash$ `copy` $(R\ x)$ N, where x is a new eigenvariable. Since instantiating an eigenvariable in a sequent with any term of the same type yields another provable sequent, then we know that $\mathcal{C},$ `copy` M $M \vdash$ `copy` $(R\ M)$ N is provable. By the previous theorem about `copy`, we also know that $\mathcal{C} \vdash$ `copy` M M holds and by the cut rule of the sequent calculus (modus ponens), we know that $\mathcal{C} \vdash$ `copy` $(R\ M)$ N is provable which means (using again the theorem about `copy`) that N is equal to $(R\ M)$.

```
kind ty       type.
type i, j     ty.                % Examples of primitive types
type arr      ty -> ty -> ty.    % The arrow type
type of       tm -> ty -> o.

of (app M N) A          :- of M (arr B A), of N B.
of (abs R)   (arr A B) :- pi x\ of x A => of (R x) B.
```

Fig. 6 A relational specification of object-level typing.

One of the keys to reasoning about relational specifications using logical specifications is the central use of sequent calculus judgments. For example, in the argument above, we did not attempt to reason by induction on the provability of $\vdash$ `copy` M N but rather on the provability of the sequent $\Gamma \vdash$ `copy` M N for suitable context Γ.

8.3 The open-world and closed-world perspectives

As previous examples have illustrated, the specification of atomic formulas, such as `of M N` and `copy M N`, assume the *open world* assumption. For example, in order to prove `copy (abs R) (abs S)` from assumptions $\mathcal{C}$, the process of searching for a proof generates a new member (an eigenvariable) of the type `tm`, say `c`, and adds the formula `copy c c` to the set of assumptions $\mathcal{C}$. Thus, we view the type `tm` and the theory (the logic program) about members of that type as expandable. Such an *open world perspective* is common in relational specification languages that manipulate λ-tree syntax [69, 99, 100, 128].

The open-world perspective to specifications has, however, a serious problem: in that setting, it is not generally possible to prove interesting negations. Figure 6 contains the λProlog specification of simple typing of untyped λ-terms. Note that the second clause in that figure encodes the formula $(*)$ in Section 5.1. Given those clauses, one would certainly want to prove that self-application in the untyped λ-calculus does not have a simple typing: for example, our metalogic should be strong enough to prove

$$\Sigma : \mathcal{P} \vdash \neg\, \exists y : ty.\ of\ (abs\ \lambda x\ (app\ x\ x))\ y,$$

where Σ is a signature (containing at least the type declarations in Figures 4 and 6) and $\mathcal{P}$ is the specification of the $(of\ \cdot\ \cdot)$ predicate in Figure 6. However, the inference rules of the intuitionistic logic principles that we have motivated so far are not strong enough to prove this negation: such a proof requires the use of induction.

The contrast to the open-world perspective is the familiar closed-world perspective. Consider proving the theorem $\forall n[\ fib(n) = n^2 \ \supset\ n \leq 20\ ]$, where $fib(n)$ is the n^{th} Fibonacci number. Of course, we do not attempt a proof by assuming the existence of a new (non-standard) natural number n for which the Fibonacci value is n^2. Instead, we prove that among the (standard) natural numbers, we find that there are only three values of n (0, 1, and 12) such that $fib(n) = n^2$ and that all three of those values are less than 20. The set of natural numbers is closed and induction allows us to prove such theorems about them.

Thus, it seems that in order to prove theorems about λ-tree syntax, we need both the open-world and the close-world perspectives: the trick is, of course, discovering how it is possible to accommodate these two conflicting perspectives at the same time.

8.4 Induction, coinduction, and λ-tree syntax

Since any formalization of metatheory needs to include induction and coinduction reasoning principles, we shall assume that these should be part of the logic we are using for reasoning. There are many ways to provide schemes for least and greatest fixed points within a proof theory setting. Gentzen used the familiar invariant-based induction rule to encode Peano arithmetic and to prove its consistency [58]. Both Schroeder-Heister [145] and Girard [62] considered sequent calculi that allowed for the unfolding of fixed point expressions but neither of them considered the problem of capturing *least* and *greatest* fixed points. Proof systems for intuitionistic and linear logics containing induction and coinduction were developed in a series of papers [14, 55, 89, 156, 160]. For the rest of this paper, we assume that the metalogic is an intuitionistic logic with inference rules for induction and coinduction. The logic $\mathcal{G}$ in [55] is an intuitionistic logic with induction and coinduction that can be used as such a metalogic. While we shall not describe the proof theory of that logic in detail here, we mention the following.

- Inductive and coinductive definitions generally need to be stratified in some manner so that cut-elimination can be proved and, as a consequence, the full logic is shown to be consistent.
- Inductive and coinductive definitions are encoded not as a theory or set of assumptions but as either auxiliary components to a sequent calculus [89, 145] or as term structures via μ- and ν-expressions [14, 17].

Given that we have adopted these strong principles in the logic, the closed-world perspective is enforced. We can recover the open-world perspective in this setting by following two steps. First, the ∇ (nabla) quantifier (described in Section 9) reintroduces the notion of generic quantification. Second, the *two-level logic* approach to reasoning (described in Section 11) allows us to embed within our (closed world) *reasoning logic* an inductive data structure which encodes the sequent calculus of the *specification logic* that permits the open world perspective.

9 The ∇-quantifier

Consider the following problem (taken from [103]) about reasoning with an object-logic. Let $\mathcal{H}$ be the set containing the following three (quantified) formulas.

$$\forall x \forall y\ (q\ x\ x\ y), \qquad \forall x \forall y\ (q\ x\ y\ x), \qquad \forall x \forall y\ (q\ y\ x\ x)$$

Here, q is a predicate constant of three arguments. The sequent

$$\mathcal{H} \vdash \forall u \forall v\ (q\ \langle u, t_1\rangle\ \langle v, t_2\rangle\ \langle v, t_3\rangle)$$

is provable (in either Gentzen's LJ or LK sequent calculus [57]) only if terms t_2 and t_3 are *equal*. If we use curly brackets to denotes the provability of object-level sequents, then this statement about object-level provability can be formalized as

$$\forall t_1 \forall t_2 \forall t_3 \big(\{\mathcal{H} \vdash \forall u \forall v [q\ \langle u, t_1\rangle\ \langle v, t_2\rangle\ \langle v, t_3\rangle]\} \supset t_2 = t_3\big).$$

Since object-level sequent calculus provability is inductively defined, one should be able to explicitly write a meta-level definition for $\{\mathcal{P} \vdash G\}$ that captures object-level

provability of the sequent $\mathcal{P} \vdash G$. When writing such a definition, one can imagine trying to treat the object-level universal quantifier as a metalevel universal quantifier, as in the following formula.

$$\forall t_1 \forall t_2 \forall t_3 \big([\forall u \forall v \{ \mathcal{H} \vdash (q\ \langle u, t_1 \rangle\ \langle v, t_2 \rangle\ \langle v, t_3 \rangle) \}] \supset t_2 = t_3 \big)$$

This second formula is only provable, however, if there are at least two different members of the underlying object-level type. That approach to proving this second formula is unfortunate since the original formula is provable without any assumptions about the inhabitants of the object-level types. Thus, it seems to be a mistake to reduce the object-level universal quantifier to the metalevel universal quantifier.

For a similar but simpler example, consider the ξ inference rule, often written as

$$\frac{t = s}{\lambda x.t = \lambda x.s}\ .$$

This inference rule violates the Perlis principle (Principle 3 in Section 4) since occurrences of x in the premise are free. If we fix this violation by inserting the universal quantifier into the rule

$$\frac{\forall x\ (t = s)}{\lambda x.t = \lambda x.s}$$

then the equivalence $\forall x\ (t = s) \equiv (\lambda x.t = \lambda x.s)$ can be proved. As argued in Section 2, this equivalence is problematic for λ-tree syntax since we want $\forall w\ \neg(\lambda x.x = \lambda x.w)$ to be provable because it is impossible for there to be a (capture-avoiding) substitution for w into $\lambda x.w$ that results in the term $\lambda x.x$. However, since this latter formula is equivalent to $\forall w\ \neg\forall x\ (x = w)$ this (first-order) formula cannot be proved since it is false for a first-order model with a singleton domain.

The ∇-quantifier [103, 104] provides an elegant logical treatment of these two examples. While this new quantifier can informally be described as providing a formalization of "newness" and "freshness" in a proof system, it is possible to describe it more formally using the mobility-of-binders theme. In particular, sequents are generalized from having one *global* signature (the familiar Σ) to also having several *local* signatures,

$$\Sigma : \sigma_1 \triangleright B_1, \ldots, \sigma_n \triangleright B_n \vdash \sigma_0 \triangleright B_0,$$

where σ_i is a list of variables, locally scoped over the formula B_i. The expression $\sigma_i \triangleright B_i$ is called a *generic judgment*. The ∇-introduction rules moves a formula-level binding to a generic judgment-level binding (when reading these proof rules from conclusion to premise).

$$\frac{\Sigma : (\sigma, x_\gamma) \triangleright B, \Gamma \vdash \mathcal{C}}{\Sigma : \sigma \triangleright \nabla x_\gamma.B, \Gamma \vdash \mathcal{C}}\ \nabla\mathcal{L} \qquad \frac{\Sigma : \Gamma \vdash (\sigma, x_\gamma) \triangleright B}{\Sigma : \Gamma \vdash \sigma \triangleright \nabla x_\gamma.B}\ \nabla\mathcal{R}$$

In these rules, the variable x is assumed to not occur in the local signature to which it is added: such an assumption is always possible since α-conversion is available for all term, formula, and sequent-level bindings. The generic judgment $(x_1, \ldots, x_n) \triangleright t = s$ can be identified, at least informally, with the generic judgment $\triangleright \nabla x_1 \cdots \nabla x_n.(t = s)$ and with the formula $\nabla x_1 \cdots \nabla x_n.(t = s)$. Since these introduction rules are the same on the left and the right, one expects that this quantifier is *self-dual*. Instead of listing

all the other inference rules for formulas using this extended sequent, we simply note that the following equivalences involving logical connectives hold as well.

$$\begin{array}{cc} \nabla x \neg Bx \equiv \neg \nabla x Bx & \nabla x(Bx \wedge Cx) \equiv \nabla x Bx \wedge \nabla x Cx \\ \nabla x(Bx \vee Cx) \equiv \nabla x Bx \vee \nabla x Cx & \nabla x(Bx \Rightarrow Cx) \equiv \nabla x Bx \Rightarrow \nabla x Cx \\ \nabla x \forall y Bxy \equiv \forall h \nabla x Bx(hx) & \nabla x \exists y Bxy \equiv \exists h \nabla x Bx(hx) \\ \nabla x \forall y Bxy \Rightarrow \forall y \nabla x Bxy & \nabla x.\top \equiv \top, \quad \nabla x.\bot \equiv \bot \end{array}$$

Note that the scope of a ∇ quantifier can be moved in over all propositional connectives. Moving it's scope below the universal and existential quantifier requires the familiar notion of *raising* [94]: that is, the when ∇ moves inside a quantified expression, the type of the quantified variable must be raised by the type of the ∇-quantified variable.

While they are formally different, the ∇-quantification is similar to the Gabbay and Pitt's freshness quantifier [50]: they are both self dual, i.e., $\nabla x \neg Bx \equiv \neg \nabla x Bx$, and in weak settings (roughly Horn clauses), they coincide [53].

The ∇-quantifier is the missing quantifier for formulating the ξ-rule: that is, the rule can now be written as

$$\frac{\nabla x\ (t = s)}{\lambda x.t = \lambda x.s}.$$

Thus, the formulas $\nabla x_1 \cdots \nabla x_n\ (t = s)$ and $(\lambda x_1 \ldots \lambda x_n.t) = (\lambda x_1 \ldots \lambda x_n.s)$ are provably equivalent. This treatment of the ξ-rule using ∇ appears to be similar to the semantic treatment of that rule using lambda algebras with indeterminates given in [149]. Using this inference rule, the following three formulas are equivalent.

$$\forall w\ \neg(\lambda x.x = \lambda x.w) \qquad \forall w\ \neg \nabla x\ (x = w) \qquad \forall w\ \nabla x\ (x \neq w).$$

Furthermore, all of these formulas are provable.

In general, ∇ does not imply $\forall$: that is, $(\nabla x.Bx) \supset (\forall x.Bx)$ is not generally provable. For example, assume that i is a primitive type and a and b are two constants of type i. The formula $\nabla x\ (x \neq a) \supset \forall x\ (x \neq a)$ is not provable since clearly $\nabla x\ (x \neq a)$ is provable while $\forall x\ (x \neq a)$ is not true. It is, however, the case that in certain settings ∇ does imply $\forall$: an important example of such a theorem is presented in Section 11.

Full details of sequent calculi involving ∇-quantification are provided elsewhere (see, [55, 104]). While we do not present the full sequent calculus rules here, we shall focus on the rules that actually complete a proof (i.e., rules with no premises). For example, the leaves of a sequent calculus proof might involve either the $\top$ on the right-hand-side or $\bot$ on the left-hand-side. There are two other possible leaves. The one involving equality would contain the generic judgment $(x_1, \ldots, x_n) \triangleright t = s$ which can be viewed as just another way to write the equality $\lambda x_1 \cdots \lambda x_n.t = \lambda x_1 \cdots \lambda x_n.s$. The final possibility involves a generalization of the initial rule: In particular, when is a sequent of the form $\Sigma : \Gamma, \sigma \triangleright A \vdash \sigma' \triangleright A'$ to be considered initial. There seems to be two natural approaches to defining the initial rule in the presence of generic judgments.

Minimal approach One approach declares $\Sigma : \Gamma, (x_1, \ldots, x_n) \triangleright A \vdash (y_1, \ldots, y_m) \triangleright A'$ to be initial exactly when $\lambda x_1 \cdots \lambda x_n.A$ and $\lambda y_1 \cdots \lambda y_m.A'$ are λ-convertible. Such a definition seems too strong, however, since the order of variables in two different local context does not seem important: in particular, it would seem natural that $\nabla x \nabla y.B$ should be logically equivalent to $\nabla y \nabla x.B$. Adopting this additional interchange principle is called the *minimal* approach and was used and analyzed in [13]. In that setting, local signature contexts are allowed to exchange the order of their variables.

Nominal approach Besides exchange, it might also seem natural to allow a form of strengthening: that is, to allow the equivalence of $\nabla x.B$ with B whenever x is not free in B. A consequence of such an equivalence is that all types are non-empty. For example, the formula $\exists x_i.B$ is not provable if the type i does not contain any inhabitants. However, the formula $\nabla y_i \exists x_i.B$ might be provable: there is, at least, one inhabitant of type i, namely, the nominal y. This kind of argument can easily be generalized to show that this strengthening equivalence implies that types for which one uses ∇ necessarily contain an infinite number of members. Baelde has argued [13] that certain adequacy issues can be complicated when strengthening is allowed and Gacek has described how to address most of those issues [52, Section 4.2]. The strengthening principle has been formally studied [54, 52, 55, 158] and implemented in the Abella theorem prover [15]. The nominal approach also allows for a different way of writing local (generic) contexts within sequents. Via the strengthening rule, all local contexts can have the same number of variables (just add more variables to those local contexts that are shorter than the local context of maximum length). Furthermore, all contexts can use the same variable names (using α-conversion). In such a setting, then, instead of writing the many local signatures that are now all the same, we can write that local signature as if it is global (although acting locally). Such a convention is taken, for example, in displaying sequents within the Abella prover.

10 The Abella proof assistant

Most of the proof theory principles and logic designs that we have motivated so far are implemented in the Abella interactive theorem prover. First implemented by Gacek in 2009 as part of his PhD [52], this prover has attracted a number of users and additional developers. Abella is written in OCaml and the most recent versions of the system are available via GitHub and OPAM [1]. A tutorial appears in [15]. The logical foundation that is closest to that which is implemented is the logic $\mathcal{G}$ in [55]. The approach to induction and coinduction in Abella differs significantly from that based on proof theory: in particular, the proof rules in $\mathcal{G}$ for induction and coinduction require explicitly providing invariants and co-invariants. However, Abella leaves such invariants implicit, opting for a more natural and convenient kind of guarded circular reasoning to account for induction and coinduction.

10.1 A simple proof using Abella

Before illustrating how Abella can deal with bindings in specifications and in reasoning, we first illustrate how to use this system to prove a few elementary theorems. Figure 7 displays the Abella specification of the natural number type, the constructors for that type, the predicate that describes the set of natural numbers, and the ternary relation that defines addition of natural numbers. Figure 8 displays the statement of three theorems that prove the commutativity of addition. Readers familiar with the Coq theorem prover will no doubt see a similarity to this style of declaration and proof script.

To illustrate the style of inductive reasoning that is used in Abella, we present some of the details in the proof of the `plus_com` theorem in Figure 8. After the statement

```
Kind nat type.
Type z nat.
Type s nat -> nat.

Define nat : nat -> prop by
  nat z ;
  nat (s N) := nat N.

Define plus : nat -> nat -> nat -> prop by
  plus z N N ;
  plus (s M) N (s K) := plus M N K.
```

Fig. 7 An Abella specification of natural numbers and addition

```
Theorem plus_zero : forall N, nat N -> plus N z N.
induction on 1. intros. case H1.
  search.
  apply IH to H2. search.

Theorem plus_succ :
        forall M N K, plus M N K -> plus M (s N) (s K).
induction on 1. intros. case H1.
  search.
  apply IH to H2. search.

Theorem plus_comm :
        forall M N K, nat K -> plus M N K -> plus N M K.
induction on 2. intros. case H2.
  apply plus_zero to H1. search.
  case H1. apply IH to H4 H3. apply plus_succ to H5. search.
```

Fig. 8 An Abella theorem file proving the commutativity of addition

of the theorem and the issuance of the `induction on 2` and `intros` proof tactics, the Abella proof state is denoted by a sequent written as follows.

```
Variables: M N K
IH : forall M N K, nat K -> plus M N K * -> plus N M K
H1 : nat K
H2 : plus M N K @
============================
 plus N M K
```

The list of variables in the first line are the eigenvariables that are bound over this sequent. The assumption `IH` is the inductive assumption and `H1` and `H2` are two additional assumptions that can be used to prove the last formula displayed. Note the addition of the annotation `*` in the inductive hypothesis and `@` in the `H2` assumption: these are used to stop (guard against) the fallacious circular reasoning that could result by applying the inductive hypothesis too quickly in the proof. In particular, inductive restrictions are represented by tagging an atomic formula with `*` (smaller) and with `@` (equal). These annotations are used to implicitly track the size of inductive arguments rather than using explicit numeric values. Experience with Abella suggests that using these annotations is much more natural than requiring the insertion of an actual invari-

ant (as required by the proof system for $\mathcal{G}$): the soundness of using such annotations is argued in [52].

To continue with this proof, one must perform a case analysis on `H2` before applying the inductive hypothesis. In particular, the `case H2` proof step results in two subcases (in both cases, the `@` annotation is replaced with a `*` annotation). The first one is represented by the proof state

```
Variables: K
IH : forall M N K, nat K -> plus M N K * -> plus N M K
H1 : nat K
============================
 plus K z K
```

This case can be proved by invoking the previously proved lemma `plus_zero`. The second case is represented by the proof state

```
Variables: N K1 M1
IH : forall M N K, nat K -> plus M N K * -> plus N M K
H1 : nat (s K1)
H3 : plus M1 N K1 *
============================
 plus N (s M1) (s K1)
```

The `case H1` command performs inversion on the `H1` assumption: that is, the only way that `(s K1)` can be a natural number is for `K1` to be a natural number: the state of the proving session is now

```
Variables: N K1 M1
IH : forall M N K, nat K -> plus M N K * -> plus N M K
H3 : plus M1 N K1 *
H4 : nat K1
============================
 plus N (s M1) (s K1)
```

The `apply IH to H4 H3` proof step employs the inductive assumption (since the `*` annotations in `IH` and `H3` match) and the `plus_succ` theorem completes the proof.

10.2 The π-calculus

The π-calculus [107, 108] is an interesting challenge for formalization since its metatheory must deal with not only bindings, substitution, and α-conversion but also with induction and coinduction. This calculus also has a mature theory [109, 144] which is a great aid in developing and judging formalizations. We shall assume that the reader is already familiar with the traditional formalization and meta-theory of the π-calculus as given in these references. In this section, we present an alternative formalization of the π-calculus that makes use of the ∇-quantifier and the logic $\mathcal{G}$.

10.2.1 Encoding the syntax of the π-calculus

In order to encode the π-calculus processes, we introduce two primitive types p and n denoting the syntactic categories for *processes* and *names*, respectively. The syntax of

```
Kind n, p          type.

Type null          p.
Type taup          p -> p.
Type plus, par     p -> p -> p.
Type match, out    n -> n -> p -> p.
Type in            n -> (n -> p) -> p.
Type nu                 (n -> p) -> p.
```

Fig. 9 Abella specifications for the syntax of the finite π-calculus.

processes for the π-calculus is generally given as follows:

$$P := 0 \mid \tau.P \mid x(y).P \mid \bar{x}y.P \mid (P \mid P) \mid (P + P) \mid (y)P \mid [x = y]P.$$

Expressions in the π-calculus can be formalized as simply typed λ-terms using the signature of constants given in Figure 9. That signature contains one constructor of type `p` for each different kind of expression allowed. In particular, there are two binding constructors: the *restriction* operator $(y)P$ is encoded using the constant `nu` of type `(n -> p) -> p` and the *input* prefix $x(y).P$ is encoded using the constant `in` of type `n -> (n -> p) -> p`. The expressions `(nu (y\ P y))` and `(in x (y\ P y))` encodes π-calculus expressions of the form $(y)P_y$ and $x(y).P_y$, respectively, where the expression P_y is a term of type `p` which may contain a free occurrence of the variable y of type `n`. Since the equality of simply typed λ-terms used here includes the η-rule, the expressions `(nu (y\ P y))` and `(in x (y\ P y))` can also be written as `(nu P)` and `(in x P)`.

Formally, this version of the π-calculus is usually referred to as the *finite* π-calculus since it lacks a replication operator or any method for recursive definitions. Versions of the π-calculus with replications or recursive definitions can be treated similarly [159].

10.2.2 Encoding the labeled transition system

In order to encode π-calculus transitions we introduce a new primitive type for the syntactic category of *action* expressions. There are three constructors for actions: $\tau : a$ for *silent* actions, $\downarrow : n \to n \to a$ for *input* actions, and $\uparrow : n \to n \to a$ for *output* actions. The action of inputting y on channel x is written as $\downarrow xy : a$ and the action of outputting y on channel x is written as $\uparrow xy : a$. The bound actions are expressions of the form $\downarrow x : n \to a$ (bound input on channel x) and $\uparrow x : n \to a$ (bound output on channel x). Constructors for encoding τ, $\uparrow$, and $\downarrow$ are given in the first lines of Figure 10.

One-step transitions for π-calculus expressions are encoded using two ternary-predicates: the arrow $\cdot \xrightarrow{\cdot} \cdot$ of type $p \to a \to p \to prop$ and the arrow $\cdot \xrightarrow{\cdot} \cdot$ of type $p \to (n \to a) \to (n \to p) \to prop$. Here, $P \xrightarrow{A} Q$ encodes a transition where A is a free or silent action and $P \xrightarrow{A} Q$ encodes a transition where A is a bound action. In particular, $P \xrightarrow{\downarrow x} M$ encodes the fact that P makes a bound input action $\downarrow x : n \to a$ to the abstracted process $M : n \to p$ and $P \xrightarrow{\uparrow x} M$ encodes the fact that P makes a bound output action, $\uparrow x : n \to a$ to the abstracted process $M : n \to p$.

The following rules are from the semantic definition of the π-calculus in [108].

$$\frac{}{\bar{x}y.P \xrightarrow{\bar{x}y} P} \qquad \frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'}\ y \notin \mathrm{n}(\alpha) \qquad \frac{P \xrightarrow{\bar{x}y} P'}{(y)P \xrightarrow{\bar{x}(w)} P'\{w/y\}}\ \begin{array}{l} y \neq x \\ w \notin fv((y)P') \end{array}$$

These inference rules—named OUTPUT-ACT, RES, and OPEN, respectively—are easily encoded using the following clauses.

```
one  (out X Y P) (up X Y) P ;
one  (nu P) A (nu P') := nabla x, one (P x) A (P' x);
oneb (nu M) (up X) P' :=
             nabla y, one (M y) (up X y) (P' y);
oneb (nu P) A (y\ nu (x\ P' x y)) :=
             nabla x, oneb (P x) A (P' x);
```

The rule called RES is encoded with two clauses since the original rule can be applied to both free and bound action transitions. Note that there is no need to encode the side conditions present in the inference rule version of these rules since the usual notion of quantifier scoping (including the ∇-quantifier) correctly captures those side conditions. Note also that the type of the bound variable `P'` in these clauses varies: in the second and fourth clauses, its type is $n \to p$ while in the third clause, its type is $n \to n \to p$. The full encoding of the one step (late) transitions for the finite π-calculus (that is, the rules from [108] except for the IDE rule) is given in Figure 10. The correctness of this encoding is discussed in [104, 156].

It is proved in [104, Theorem 7.10] that when implications (and, hence, negations) are not present in the body of a specification, then occurrence of ∇ in that specification can be replaced with $\forall$ (and vice-versus) and the same atomic formulas are provable. Thus, if we are only interested in proving atomic formulas, then all of the ∇-quantifiers in Figure 10 can be changed to $\forall$ and there would be no difference between those two definitions as to the `one` and `onep` atomic formula that are provable. Specifications of the π-calculus containing universal quantifiers instead have been written in, say, λProlog [99, 102].

Consider using this transition definition in the context of a negation. The process $(y)[x = y]\bar{x}w.0$, where w is some constant, cannot make a (free or bound) transition since y has to be "new" and cannot be equal to x. Thus the following formula is provable using the specification in Figure 10.

$$\forall x \forall Q \forall \alpha.[((y)[x = y](\bar{x}w.0) \xrightarrow{\alpha} Q) \supset \bot]$$

This theorem and its brief proof can be entered into Abella as follows (here, `w` is a constant of type `name`):

```
Theorem example1 : forall x Q A,
  one (nu y\ match x y (out x w null)) A Q -> false.
  intros.  case H1. case H2.
```

The last step in this proof formally involves a unification that is rather similar to the unification problem mentioned in Section 9: that is, this last step in the proof succeeds because the unification problem $(\lambda y.x) = (\lambda y.y)$ fails (since there is no substitution for the free variable x that makes these terms equal).

```
Kind act          type.
Type tau          act.
Type up, dn       n -> n -> act.

Define
  one  : p ->       act  ->       p  -> prop,
  oneb : p -> (n -> act) -> (n -> p) -> prop
by
  oneb (in X M)    (dn X)   M ;
  one  (out X Y P) (up X Y) P ;
  one  (taup P)    tau      P ;
  one  (match X X P) A Q := one  P A Q ;
  oneb (match X X P) A M := oneb P A M ;
  one  (plus P Q) A R    := one  P A R ;
  one  (plus P Q) A R    := one  Q A R ;
  oneb (plus P Q) A M    := oneb P A M ;
  oneb (plus P Q) A M    := oneb Q A M ;
  one  (par P Q) A (par P' Q)       := one  P A P' ;
  one  (par P Q) A (par P Q')       := one  Q A Q' ;
  oneb (par P Q) A (x\ par (M x) Q) := oneb P A M ;
  oneb (par P Q) A (x\ par P (N x)) := oneb Q A N ;
  one  (nu P) A (nu Q)             := nabla x, one  (P x) A (Q x);
  oneb (nu P) A (y\ nu (x\ Q x y)) := nabla x, oneb (P x) A (Q x);
  oneb (nu M) (up X) N := nabla y, one (M y) (up X y) (N y) ;
  one (par P Q) tau (nu y\ par (M y) (N y)) :=
    exists X, oneb P (dn X) M /\ oneb Q (up X) N ;
  one (par P Q) tau (nu y\ par (M y) (N y)) :=
    exists X, oneb P (up X) M /\ oneb Q (dn X) N ;
  one (par P Q) tau (par (M Y) T) :=
    exists X, oneb P (dn X) M /\ one Q (up X Y) T ;
  one (par P Q) tau (par R (M Y)) :=
    exists X, oneb Q (dn X) M /\ one P (up X Y) R.
```

Fig. 10 Specifications of the one step (late) transitions for the finite π-calculus.

10.2.3 Some of the metatheory of the π-calculus

The quality of this specification of the π-calculus transition semantics is high, as illustrated by the following observations.

When instantiating the quantifiers (implicitly) quantifying the many clauses in Figure 10, the resulting formulas may contain subterms that are not β-normal. Only the last two clause rules yield instances in which the resulting β-redexes are actually not β_0-redexes (Section 5). If we delete those last two clauses from Figure 10, we get the π_I-calculus (the π-calculus with *internal mobility* [143]). Since the only β-conversion needed is β_0-conversion, we can view the π_I-calculus as a subset of the π-calculus in which β-conversion is only used to provide binder mobility (in the sense described in Section 5) and not more.

The definition of simulation for the π-calculus can be given as the greatest fixed point of the following recursive definition.

$$
\begin{aligned}
sim\ P\ Q \triangleq\ & \forall A, P'\,[P \xrightarrow{A} P' \Rightarrow \exists Q'.Q \xrightarrow{A} Q' \wedge sim\ P'\ Q'] \wedge \\
& \forall X, P'\,[P \xrightarrow{\downarrow X} P' \Rightarrow \exists Q'.Q \xrightarrow{\downarrow X} Q' \wedge \forall w.sim\ (P'w)\ (Q'w)] \wedge \\
& \forall X, P'\,[P \xrightarrow{\uparrow X} P' \Rightarrow \exists Q'.Q \xrightarrow{\uparrow X} Q' \wedge \nabla w.sim\ (P'w)\ (Q'w)]
\end{aligned}
$$

```
CoDefine bisim : p -> p -> prop by
  bisim P Q :=
   (forall A P', one P A P' ->
      exists Q', one Q A Q' /\ bisim P' Q') /\
   (forall X M, oneb P (dn X) M ->
      exists N, oneb Q (dn X) N /\ forall W, bisim (M W) (N W)) /\
   (forall X M, oneb P (up X) M ->
      exists N, oneb Q (up X) N /\  nabla w, bisim (M w) (N w)) /\
   (forall A Q', one Q A Q' ->
      exists P', one P A P' /\ bisim P' Q') /\
   (forall X N, oneb Q (dn X) N ->
     exists M,  oneb P (dn X) M /\ forall W, bisim (M W) (N W)) /\
   (forall X N, oneb Q (up X) N ->
     exists M,  oneb P (up X) M /\  nabla w, bisim (M w) (N w)).
```

Fig. 11 Bisimulation for the π-calculus

Bound actions use two different quantifiers to handle the continuation of a process: an input bound action uses $\forall$ while the output bound action uses ∇. The corresponding coinductive definition of bisimulation is displayed using Abella syntax in Figure 11. As proved in [104], this coinductive definition corresponds to *open* bisimulation on the finite π-calculus. If we changed from the intuitionistic logic of Abella to classical logic instead, then *late* bisimulation is captured. In particular, late bisimulation can be captured by assuming $\forall x \forall y.\ x = y \vee x \neq y$, a particular instance of the excluded middle [159].

The specification for the π-calculus above (and some variants of it) has also been used in formal proofs of various aspects of the metatheory of the π-calculus. For example, the Abella website [1] contains a formal proof that open bisimulation is a congruence and that bisimulation-up-to bisimilarity [106,139] is sound (see also [27]). The modal logics for π-calculus given in [109] have also been formalized in the logic underlying Abella [4,161,159]. The model checker Bedwyr [16] provides an automated implementation of part of Abella: that implementation provides a basic model checker for the π-calculus [157] and some extensions of it [162].

The Coq theorem prover has been used to formalize the metatheory of the π-calculus. Generally, two kinds of packages have been added to Coq for this purpose. First, a package that provides flexible methods for doing coinduction following, say, the Knaster-Tarski fixed point theorems, is necessary. Indeed, such a package has been implemented and used to prove various metatheorems surrounding bisimulation-up-to (including the subtle metatheory surrounding weak bisimulation) [22,137,138]. Second, a package for the treatment of bindings and names that are used to describe the operational semantics of the π-calculus. Such packages exist (for example, see [10]) and, when combined with treatments of coinduction, may allow one to make progress on the metatheory of the π-calculus. Recently, the Hybrid systems [44] has shown a different way to incorporate both induction, coinduction, and binding into a Coq (and Isabelle) implementation. Such an approach could be seen as one way to implement this metatheory task on top of an establish formalization of mathematics. A still different approach to using Coq was taken by Honsell, Miculan, and Scagnetto in [72] in their "Theory of Context." While their logical foundations is rather different from that described here, their specifications of, say, bisimulation strongly resemble the specifications given above. In particular, the ∇-quantifier can be used to define in Abella a

```
Kind   atm, atmlist       type.

Type   nl                 atmlist.
Type   cons               atm -> atmlist -> atmlist.

Type   tt                 o.
Type   atom               atm -> o.
Type   or, and            o -> o -> o.
Type   imp                atm -> o -> o.
Type   all                (tm -> o) -> o.
Type   ex                 (tm -> o) -> o.

Define seq : nat -> atmlist -> o -> prop by
  seq (s N) L tt ;
  seq (s N) L (atom A)  := mem A L ;
  seq (s N) L (or A B)  := seq N L A ;
  seq (s N) L (or A B)  := seq N L B ;
  seq (s N) L (and A B) := seq N L A /\ seq N L B ;
  seq (s N) L (imp A B) := seq N (cons A L) B ;
  seq (s N) L (all G)   := nabla x,  seq N L (G x) ;
  seq (s N) L (ex G)    := exists X, seq N L (G X) ;
  seq (s N) L (atom A)  := exists B, prog A B /\ seq N L B.
```

Fig. 12 Definition of provability in the specification level logic.

freshness predicate *fresh* x t which holds if x is a nominal that does not appear free in t. If we let B be a formula whose free variables are among $z, x_1, \ldots, x_n$, and let $\mathbf{x}$ be the term $x_1 :: \ldots :: x_n :: nil$ (where :: and nil are constructors for lists), then the three formulas

$$\nabla z.B \qquad \exists z.(\mathit{fresh}\ z\ \mathbf{x} \wedge B) \qquad \forall z.(\mathit{fresh}\ z\ \mathbf{x} \supset B)$$

are provably equivalent in Abella [55]. Replacing the freshness predicate with the `notin` predicate of [72] illustrates the strong similarity between their encoding of bisimulation and the one in Figure 11.

11 The two-level logic approach as implemented in Abella

At the end of Section 8, the need for both the open-world assumption (for a declarative treatment of λ-tree syntax) and the closed-world assumption (for the treatment of induction) was motivated. Following the two-level logic approach of [56,90], we now present an Abella (reasoning-level logic) specification of provability for an object-level logic (essentially a restricted form of λProlog). This design will formally allow establishing "substitution lemmas for free" by using cut-elimination.

Consider the specification of an (object) logic (terms of type `o`) as well as of cut-free provability via the `seq` predicate. In this example, object-level formulas are limited so that the left of an implication is restricted to be only an atomic formula. This restriction is sometimes used to make the formal presentation of object-level provability simpler [88,90]: the recent versions of Abella, however, removes that restriction [15, 171]. This specification is also partial since it does not define the (obvious) membership relation `mem` on `atmlist` nor the `prog` relationship. This latter relation is used to hold the logic programming specification on which we plan to prove theorems. For example,

```
Theorem seq_monotone : forall L1 L2 G N,
  seq N L1 G -> (forall X, mem X L1 -> mem X L2) -> seq N L2 G.

Theorem seq_cut : forall K L G N M,
  nat N -> seq N (cons K L) G -> nat M -> seq M L (atom K) ->
    exists P, nat P /\ seq P L G.
```

Fig. 13 Some metatheorems which have been proved about the specification level logic.

```
Theorem mem_inst : forall L A,
   nabla (x:tm), mem (A x) (L x) -> forall T, mem (A T) (L T).

Theorem seq_inst : forall N L G,
   nabla (x:tm), seq N (L x) (G x) -> forall T, seq N (L T) (G T).
```

Fig. 14 Two theorems of the form $\nabla x.P(x) \supset \forall x.P(x)$.

the evaluation and typing rules for the untyped λ-calculus given in Figures 4 and 6 can be written as the following definition of `prog` within Abella. (Essentially, `prog` is the prefix version of what is written as the infix symbol `:-` in λProlog.)

```
Define prog : atm -> o -> prop by
  prog (eval (abs R) (abs R)) tt ;
  prog (eval (app P Q) V) (and (atom (eval P (abs R)))
                               (atom (eval (R Q) V))) ;
  prog (of (abs R) (arrow A B))
       (all x\ imp (of x A) (atom (of (R x) B))) ;
  prog (of (app P Q) B) (and (atom (of P (arrow A B)))
                             (atom (of Q A))).
```

For the sake of providing variety, this definition of `eval` encodes call-by-name evaluation whereas the specification in Figure 4 encodes call-by-value evaluation.

The two theorems in Figure 13 state properties of object-level provability. Theorem `seq_monotone` states that when every member of the list of hypotheses `L1` is a member of the list of hypotheses `L2` then the existence of a proof of height bounded by `N` using `L1` guarantees the existence of a proof of height bounded by `N` using `L2`. This theorem can be used to show that contraction and weakening are admissible rules of inference. Theorem `seq_cut` states that atomic instances of the cuts rule are admissible.

As noted in Section 9, it is not generally the case that $\nabla x.P(x) \supset \forall x.P(x)$ is provable. There are a number of situations, however, where this entailment does hold. For example, Theorem `mem_inst` in Figure 13 states that this entailment holds when the property $P(x)$ refers to membership in a list and Theorem `seq_inst` states that this also holds for object-level provability. Both of these theorems can be proved by straightforward induction.

Given the theorems above regarding properties of the object-logic, we can now formally prove the type-preservation theorem that was informally proved in Section 6.4. Given the encoding described above, this can be written as

```
Theorem type_preserve : forall E V A M N N',
  nat N' -> lt N N' -> seq N nl (atom (eval E V)) ->
  nat M  ->            seq M nl (atom (of E A))   ->
    exists P, nat P /\ seq P nl (atom (of V A)).
```

Although this encoding is a bit awkward to read, the formal proof in Abella of this theorem follows the proof outline given in Section 6.4: where an eigenvariable is instantiated, the theorem `seq_inst` is invoked and where modus ponens was used, the theorem `seq_cut` is invoked.

Abella provides object-level provability as a built-in feature. In particular, the Abella expression `{L |- G}` abbreviates `exists N, nat N /\ seq N L G`: if the list `L` is empty, then that expression is written simply as `{G}`. The three theorems `seq_monotone`, `seq_cut`, and `seq_inst` are made available as specific tactics within Abella. Furthermore, Abella can build the `prog` predicate from (restricted) λProlog logic program and signature files.

To illustrate these features within Abella, assume that the clauses in Figures 4 and 6 are gathered together into one logic program that is loaded into Abella (via the `prog` binary predicate). The `type_preserve` theorem above can be rewritten and formally proved correct using the following.

```
Theorem type_preserve :
  forall E V T, {eval E V} -> {of E T} -> {of V T}.
induction on 1. intros. case H1.
  search.
  case H2.
     apply IH to H3 H6. case H8. apply IH to H4 H7.
     inst H9 with n1 = U. cut H11 with H10.
     apply IH to H5 H12. search.
```

Just before the `inst` command is issued, the proof system of Abella appears as follows.

```
Variables: V T U R N M B
IH : forall E V T, {eval E V}* -> {of E T} -> {of V T}
H3 : {eval M (abs R)}*
H4 : {eval N U}*
H5 : {eval (R U) V}*
H6 : {of M (arr B T)}
H7 : {of N B}
H9 : {of n1 B |- of (R n1) T}
H10 : {of U B}
============================
 {of V T}
```

The list of variables (in the first line) are the eigenvariables that are bound in this sequent. The inductive hypothesis is labeled with `IH` and the asterisks on some of the assumptions are part of Abella's approach to doing induction (as mentioned in Section 10.1: see [15,52] for more on this approach to induction). Assumption `H9` captures the object-level provability judgment that for a fresh object-level eigenvariable `n1` (captured as a nominal), that the object-level sequent `{of n1 B |- of (R n1) T}` is provable. The `inst H9 with n1 = U` is responsible for instantiating the nominal `n1` with the term `U` yielding the hypothesis

```
H11 : {of U B |- of (R U) T}
```

That is, since `H9` holds generically (that is, for a nominal `n1`) then it holds for every instant of that nominal. Similarly, the `cut` command applies that hypothetical to the assumption `H10` and this yields the additional assumption

```
H12 : {of (R U) T}
```

Applying the inductive hypothesis `IH` to hypotheses `H5` and `H12` finally yields the desired goal. The combination of the `inst` and `cut` commands provides an elegant and completely formal proof of the key *substitution lemma* in the proof, namely that if the type of `(abs R)` is the arrow type `(arr B T)` and if `U` has type `B` then the result of instantiating the abstract `(abs R)` with `U`, that is, `(R U)`, has type `T`. The proof of this substitution lemma follows the simple line of reasoning described in Section 6.4.

The Abella system has been successfully used to prove a range of metatheoretic properties about well known formal systems. Several such examples are listed below: the formal treatment of several of these topics can be found on the Abella prover's website [1].

- Untyped λ-calculus: Takahashi's proof of the Church-Rosser property using complete developments, a characterization of β-reduction via paths through terms; Loader's proof of standardization; type preservation of call-by-name and call-by-value for simple types and system F types; and Huet's proof of the cube property of λ-calculus residuals [3].
- Simply typed λ-calculus: Tait's logical relations argument for weak normalization and Girard's proof of strong normalization.
- Object-level proof systems: cut-elimination and the completeness of a Frege-style proof system.
- Formalized metatheory for the process calculi CCS and π-calculus: see Section 10.2 and [4,159].
- Specifications and correctness proofs for various techniques used by compilers for functional programming languages [170,172].

12 Related work

Two broad avenues of attacking the general area of mechanizing metatheory have been developed. One such approach has involved designing and implementing computer systems that provide tools for analyzing and automating entire programming languages and specification languages. While there are too many such systems to survey them here, we mention a few from different periods during the last 3 decades. Two systems from Inria in the 1980's were the Mentor [38] and Centaur systems [24]. The Ergo Support System [81] was also from that same period: that system proposed to use the Elf automation of LF signatures for capturing programming language specifications. The following systems have been built to support reasoning about various concurrency calculi: the Concurrency Workbench [32], the Mobility Workbench [169], and the more recent Psi-calculus Workbench [23]. A number of recent textbooks have been written that use the Agda and Coq proof assistants to formally reason about programs and programming languages [30,130,154]. The OTT system [150] provides a convenient means for defining the static and dynamic semantics of programming languages and for exporting those definitions to various proof assistants.

A second approach to mechanizing metatheory has been to look into the foundations of logic—particularly following the perspectives found in proof theory and type theory—and uncover logical principles that can be directly employed to support mechanized metatheory. As we have illustrated here, pursuing this course has occasionally lead researchers to develop new logical principles (for example, λ-tree syntax) and

connectives (for example, the ∇-quantifier). This line of research can be traced back to Church's Simple Theory of Types [31] since it contains the main ingredients for supporting λ-tree syntax. Because Church was interested in supporting mathematical reasoning, his addition of mathematical axioms (extensionally and choice, in particular) made the theory of λ-expressions in the resulting system too strong to support λ-tree syntax: the weaker Elementary Type Theory (ETT) does provide, however, a good starting point. Probably the first computational system that captured some aspects of λ-tree syntax was the template-based rewriting system of Huet & Lang [76] in which second-order matching was used to match and rewrite program expressions containing bindings. λProlog was the first programming language to support the full range of features needed to manipulate λ-tree syntax [97,98,101] and the Elf system [125] provided similar support by automating proof search based on LF dependently-typed λ-calculus.

Shortly after these first programming language systems appeared, tools for performing inductive proofs over λ-tree syntax were implemented. The first such system was Twelf [128], an extension to the Elf system that could determine, for example, that given dependently-typed relations were total and/or functional. A proof theory for induction (over natural numbers) and an early implementations of the two-level logic approach to reasoning based on the Pi proof editor [41] are given in [87–89]. With the addition of the ∇-quantifier [103,104] and stronger forms of induction and coinduction [14,54,160], the $\mathcal{G}$ logic was design to incorporate these various features: it is that logic which is built into the Abella theorem prover. Besides the Abella and Twelf system, a number of other implemented systems support some or all aspects of λ-tree syntax: these include Beluga [129], Hybrid [44], Isabelle [123], Minlog [147], and the Teyjus [115] and ELPI [40] implementations of λProlog. Some of these systems have been explicitly compared and contrasted in recent papers [45,46,78,112]. Additionally, benchmark problems that are unique to metaprogramming problems have been proposed to test the ability of mechanized metatheory provers [12,47].

There is a spectrum of how abstract or concrete the encoding of bindings in syntax can be in different computer systems. The λ-tree syntax approach is at the abstract extreme since it hides away completely the names of bindings and it allows term-level binding to move into bindings of the surrounding proof state. Techniques that encode bindings with string names are, in many ways, too concrete. The use of de Bruijn's nameless dummies [36] provides some abstraction but often many concrete details remain to clutter up meaningful semantic specifications. Intermediate approaches are also possible: for example, the nominal logic approach of Pitts and Gabbay [131,49], which abstracts away variable names without using the notion of binder mobility, has successfully been attached to a number of programming languages, logics, and theorem provers [10,28,132,136,164–166].

13 Conclusions

We have argued that parsing concrete syntax into parse trees does not yield a sufficiently abstract representation of expressions and we have motivated the λ-tree syntax approach for treating binders more abstractly. For a programming language or proof assistant to support this level of abstraction in syntax, equality of syntax must be based on α and at least the β_0 subset of β conversion and must allow for the mobility of binders from within terms to within formulas (i.e., quantifiers) and proof state (i.e.,

eigenvariables). We have also argued that the logic programming paradigm—broadly interpreted—provides an elegant and high-level framework for specifying both computation and deduction involving syntax containing bindings. This framework is offered up as an alternative to the more conventional approaches to mechanizing metatheory using formalizations based on more conventional mathematical concepts. While the POPLmark challenge was based on the assumption that increments to existing provers will solve the problems surrounding the mechanization of metatheory, we have argued against that assumption.

We have described an extension of ETT targeting metatheory and not mathematics. The resulting logic provides for λ-tree syntax in a direct fashion, via binder-mobility, ∇-quantification, and the unification of λ-terms. Induction over syntax containing bindings is available: in its richest setting, such induction is done over sequent calculus proofs of typing derivations. The Abella interactive theorem prover, which includes these logical principles, has been used to capture important aspects of the metatheory of the λ-calculus, π-calculus, programming languages, and object-logics.

The shift from conventional proof assistants based on functional programming principles to assistants based on logic programming principles does disrupt a number of aspects of proof assistants. For example, when computations are naturally considered as functional, it seems that there is a loss of expressiveness and effectiveness if one must write those specifications using relations. Recent work shows, however, that when a relation actually encodes a function, it is possible to use the proof search framework to actually compute that function [59]. A popular feature of many proof assistants is the use of tactics and tacticals, which have been implemented using functional programs since their introduction [65]. There are good arguments, however, that those operators can be given elegant and natural implementations using (higher-order) logic programs [39, 42, 99]. We have tried to argue in this paper that the disruptions that result from such a shift are well worth exploring.

Finally, we have argued that basic aspects of provers—terms and equality on them—need to be rethought and re-implemented in order to build a new approach to proving. During the past 30 years, a number of researchers have been working on developing the theoretical background and related implementations to help validate this new approach to theorem proving of metatheory. Two such computer systems, λProlog and Abella, are highlighted.

Acknowledgments. I thank Gopalan Nadathur and the anonymous reviewers for their many helpful comments on an earlier draft of this paper. This work was funded in part by the ERC Advanced Grant ProofCert.

References

1. The Abella prover, 2012. Available at `http://abella-prover.org/`.
2. S. Abramsky. The lazy lambda calculus. In D. A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Welsey, Reading, MA, 1990.
3. Beniamino Accattoli. Proof pearl: Abella formalization of lambda calculus cube property. In Chris Hawblitzel and Dale Miller, editors, *Second International Conference on Certified Programs and Proofs*, volume 7679 of *LNCS*, pages 173–187. Springer, 2012.
4. Ki Yung Ahn, Ross Horne, and Alwen Tiu. A Characterisation of Open Bisimilarity using an Intuitionistic Modal Logic. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory (CONCUR 2017)*, volume 85 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 7:1–7:17, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

5. Thorsten Altenkirch. A formalization of the strong normalization proof for system F in LEGO. In *Typed Lambda Calculi and Applications (TLCA)*, volume 664, pages 13–28, 1993.
6. Peter B. Andrews. Resolution in type theory. *J. of Symbolic Logic*, 36:414–432, 1971.
7. Peter B. Andrews. Provability in elementary type theory. *Zeitschrift fur Mathematische Logic und Grundlagen der Mathematik*, 20:411–418, 1974.
8. Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Kluwer Academic Publishers, second edition, 2002.
9. Andrew W. Appel and Amy P. Felty. Polymorphic lemmas and definitions in λProlog and Twelf. *Theory and Practice of Logic Programming*, 4(1-2):1–39, 2004.
10. Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. Engineering formal metatheory. In *35th ACM Symp. on Principles of Programming Languages*, pages 3–15. ACM, January 2008.
11. Brian Aydemir, Stephan A. Zdancewic, and Stephanie Weirich. Abstracting syntax. Technical Report MS-CIS-09-06, University of Pennsylvania, 2009.
12. Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In *Theorem Proving in Higher Order Logics: 18th International Conference*, number 3603 in LNCS, pages 50–65. Springer, 2005.
13. David Baelde. On the expressivity of minimal generic quantification. In A. Abel and C. Urban, editors, *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP 2008)*, number 228 in ENTCS, pages 3–19, 2008.
14. David Baelde. Least and greatest fixed points in linear logic. *ACM Trans. on Computational Logic*, 13(1), 2:1-2:44, April 2012.
15. David Baelde, Kaustuv Chaudhuri, Andrew Gacek, Dale Miller, Gopalan Nadathur, Alwen Tiu, and Yuting Wang. Abella: A system for reasoning about relational specifications. *Journal of Formalized Reasoning*, 7(2), 2014.
16. David Baelde, Andrew Gacek, Dale Miller, Gopalan Nadathur, and Alwen Tiu. The Bedwyr system for model checking over syntactic expressions. In F. Pfenning, editor, *21th Conf. on Automated Deduction (CADE)*, number 4603 in LNAI, pages 391–397, New York, 2007. Springer.
17. David Baelde and Dale Miller. Least and greatest fixed points in linear logic. In N. Dershowitz and A. Voronkov, editors, *International Conference on Logic for Programming and Automated Reasoning (LPAR)*, volume 4790 of *LNCS*, pages 92–106, 2007.
18. Henk P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, April 1991.
19. Christoph Benzmüller and Dale Miller. Automation of higher-order logic. In J. Siekmann, editor, *Computational Logic*, volume 9 of *Handbook of the History of Logic*, pages 215–254. North Holland, 2014.
20. Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.
21. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
22. Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 457–468. ACM, 2013.
23. Johannes Borgstrom, Ramūnas Gutkovas, Ioana Rodhe, and Björn Victor. The psi-calculi workbench: A generic tool for applied process calculi. *ACM Trans. Embed. Comput. Syst.*, 14(1):9:1–9:25, January 2015.
24. P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Third Annual Symposium on Software Development Environments (SDE3)*, pages 14–24, Boston, 1988.
25. Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda - A functional language with dependent types. In *TPHOLs*, volume 5674, pages 73–78. Springer, 2009.
26. Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pages 1–46, May 2011.
27. Kaustuv Chaudhuri, Matteo Cimini, and Dale Miller. A lightweight formalization of the metatheory of bisimulation-up-to. In Xavier Leroy and Alwen Tiu, editors, *Proceedings of the 4th ACM-SIGPLAN Conference on Certified Programs and Proofs*, pages 157–166, Mumbai, India, January 2015. ACM.

28. James Cheney and Christian Urban. Alpha-Prolog: A logic programming language with names, binding, and alpha-equivalence. In Bart Demoen and Vladimir Lifschitz, editors, *Logic Programming, 20th International Conference*, volume 3132 of *LNCS*, pages 269–283. Springer, 2004.
29. Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 143–156. ACM, 2008.
30. Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.
31. Alonzo Church. A formulation of the Simple Theory of Types. *J. of Symbolic Logic*, 5:56–68, 1940.
32. Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):36–72, 1993.
33. Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoët, and Gilles Kahn. Natural semantics on the computer. Research Report 416, INRIA, Rocquencourt, France, June 1985.
34. Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
35. Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, February/March 1988.
36. Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with an application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5):381–392, 1972.
37. Joëlle Despeyroux, Amy Felty, and Andre Hirschowitz. Higher-order abstract syntax in Coq. In *Second International Conference on Typed Lambda Calculi and Applications*, pages 124–138, April 1995.
38. Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang. Programming environments based on structured editors: The MENTOR experience. Technical report, Inria, 1980.
39. Cvetan Dunchev, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing HOL in an higher order logic programming language. In Gilles Dowek, Daniel R. Licata, and Sandra Alves, editors, *Proceedings of the Eleventh Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP 2016, Porto, Portugal, June 23, 2016*, pages 4:1–4:10. ACM, 2016.
40. Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λProlog interpreter. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *LNCS*, pages 460–468. Springer, 2015.
41. Lars-Henrik Eriksson. Pi: an interactive derivation editor for the calculus of partial inductive definitions. In A. Bundy, editor, *Proceedings of the Twelfth International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 821–825. Springer, June 1994.
42. Amy Felty and Dale Miller. Specifying theorem provers in a higher-order logic programming language. In *Ninth International Conference on Automated Deduction*, number 310 in LNCS, pages 61–80, Argonne, IL, May 1988. Springer.
43. Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In Mark Stickel, editor, *Proceedings of the 1990 Conference on Automated Deduction*, volume 449 of *LNAI*, pages 221–235. Springer, 1990.
44. Amy Felty and Alberto Momigliano. Hybrid: A definitional two-level approach to reasoning with higher-order abstract syntax. *J. of Automated Reasoning*, 48:43–105, 2012.
45. Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 1–A common infrastructure for benchmarks. Technical report, Arxiv, 2015.
46. Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. The next 700 challenge problems for reasoning with higher-order abstract syntax representations: Part 2–A survey. *J. of Automated Reasoning*, 55(4):307–372, 2015.
47. Amy P. Felty, Alberto Momigliano, and Brigitte Pientka. Benchmarks for reasoning with syntax trees containing binders and contexts of assumptions. *Mathematical Structures in Computer Science*, 28:1507-1540, 2017.

48. M. P. Fiore, G. D. Plotkin, and D. Turi. Abstract syntax and variable binding. In *14th Symp. on Logic in Computer Science*, pages 193–202. IEEE Computer Society Press, 1999.
49. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Symp. on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, 1999.
50. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.
51. Andrew Gacek. The Abella interactive theorem prover (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Fourth International Joint Conference on Automated Reasoning*, volume 5195 of *LNCS*, pages 154–161. Springer, 2008.
52. Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems*. PhD thesis, University of Minnesota, 2009.
53. Andrew Gacek. Relating nominal and higher-order abstract syntax specifications. In *Proceedings of the 2010 Symposium on Principles and Practice of Declarative Programming*, pages 177–186. ACM, July 2010.
54. Andrew Gacek, Dale Miller, and Gopalan Nadathur. Combining generic judgments with recursive definitions. In F. Pfenning, editor, *23th Symp. on Logic in Computer Science*, pages 33–44. IEEE Computer Society Press, 2008.
55. Andrew Gacek, Dale Miller, and Gopalan Nadathur. Nominal abstraction. *Information and Computation*, 209(1):48–73, 2011.
56. Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *J. of Automated Reasoning*, 49(2):241–273, 2012.
57. Gerhard Gentzen. Investigations into logical deduction. In M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131. North-Holland, Amsterdam, 1935.
58. Gerhard Gentzen. New version of the consistency proof for elementary number theory. In M. E. Szabo, editor, *Collected Papers of Gerhard Gentzen*, pages 252–286. North-Holland, Amsterdam, 1938. Originally published 1938.
59. Ulysse Gérard and Dale Miller. Separating functional computation from relations. In Valentin Goranko and Mads Dam, editors, *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*, volume 82 of *LIPIcs*, pages 23:1–23:17, 2017.
60. Ulysse Gérard and Dale Miller. Functional programming with λ-tree syntax: a progress report. In *13th international Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, Oxford, United Kingdom, July 2018.
61. Jean-Yves Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, Amsterdam, 1971.
62. Jean-Yves Girard. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu, February 1992.
63. Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte der Mathematischen Physik*, 38:173–198, 1931. English Version in [167].
64. M. J. C. Gordon and T. F. Melham. *Introduction to HOL – A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
65. Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.
66. Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, pages 169–186. MIT Press, 2000.
67. John Hannan. Extended natural semantics. *J. of Functional Programming*, 3(2):123–152, April 1993.
68. John Hannan and Dale Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.
69. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
70. John Harrison. HOL light: an overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer, 2009.
71. Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Symp. on Logic in Computer Science*, pages 204–213. IEEE Computer Society Press, 1999.

72. Furio Honsell, Marino Miculan, and Ivan Scagnetto. π-calculus in (co)inductive type theories. *Theoretical Computer Science*, 2(253):239–285, 2001.
73. Douglas J. Howe. Proving congruence of bisimulation in functional programming languages. *Information and Computation*, 124(2):103–112, 1996.
74. Gérard Huet. The undecidability of unification in third order logic. *Information and Control*, 22:257–267, 1973.
75. Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.
76. Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.
77. Gilles Kahn. Natural semantics. In Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing, editors, *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, volume 247 of *LNCS*, pages 22–39. Springer, March 1987.
78. Jonas Kaiser, Brigitte Pientka, and Gert Smolka. Relating system F and $\lambda 2$: A case study in Coq, Abella and Beluga. In Dale Miller, editor, *FSCD 2017 - 1st International Conference on Formal Structures for Computation and Deduction*, pages 21:1–21:19, Oxford, UK, September 2017.
79. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles (22nd SOSP'09), Operating Systems Review (OSR)*, pages 207–220, Big Sky, MT, October 2009. ACM SIGOPS.
80. Ulrich Kohlenbach and Paulo Oliva. Proof mining: a systematic way of analysing proofs in mathematics. *Proceedings of the Steklov Institute of Mathematics*, 242:136–164, 2003.
81. Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis. The Ergo Support System: An integrated set of tools for prototyping integrated environments. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 25–34. ACM Press, November 1988.
82. Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
83. Chuck Liang, Gopalan Nadathur, and Xiaochu Qi. Choices in representing and reduction strategies for lambda terms in intensional contexts. *Journal of Automated Reasoning*, 33:89–132, 2005.
84. Donald MacKenzie. *Mechanizing Proof*. MIT Press, 2001.
85. Petar Maksimović and Alan Schmitt. HOCore in coq. In *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, number 9236 in LNCS, pages 278–293. Springer, 2015.
86. Per Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory Lecture Notes. Bibliopolis, Napoli, 1984.
87. Raymond McDowell. *Reasoning in a Logic with Definitions and Induction*. PhD thesis, University of Pennsylvania, December 1997.
88. Raymond McDowell and Dale Miller. A logic for reasoning with higher-order abstract syntax. In Glynn Winskel, editor, *12th Symp. on Logic in Computer Science*, pages 434–445, Warsaw, Poland, July 1997. IEEE Computer Society Press.
89. Raymond McDowell and Dale Miller. Cut-elimination for a logic with definitions and induction. *Theoretical Computer Science*, 232:91–119, 2000.
90. Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Trans. on Computational Logic*, 3(1):80–136, 2002.
91. Dale Miller. A compact representation of proofs. *Studia Logica*, 46(4):347–370, 1987.
92. Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. of Logic and Computation*, 1(4):497–536, 1991.
93. Dale Miller. Abstract syntax and logic programming. In *Logic Programming: Proceedings of the First Russian Conference on Logic Programming, 14-18 September 1990*, number 592 in LNAI, pages 322–337. Springer, 1992.
94. Dale Miller. Unification under a mixed prefix. *Journal of Symbolic Computation*, 14(4):321–358, 1992.
95. Dale Miller. Bindings, mobility of bindings, and the ∇-quantifier. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *18th International Conference on Computer Science Logic (CSL) 2004*, volume 3210 of *LNCS*, page 24, 2004.

96. Dale Miller. Finding unity in computational logic. In *Proceedings of the 2010 ACM-BCS Visions of Computer Science Conference*, ACM-BCS '10, pages 3:1–3:13. British Computer Society, April 2010.
97. Dale Miller and Gopalan Nadathur. Higher-order logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Logic Programming Conference*, pages 448–462, London, June 1986.
98. Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In Seif Haridi, editor, *IEEE Symposium on Logic Programming*, pages 379–388, San Francisco, September 1987.
99. Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.
100. Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51(1–2):125–157, 1991.
101. Dale Miller, Gopalan Nadathur, and Andre Scedrov. Hereditary Harrop formulas and uniform proof systems. In David Gries, editor, *2nd Symp. on Logic in Computer Science*, pages 98–105, Ithaca, NY, June 1987.
102. Dale Miller and Catuscia Palamidessi. Foundational aspects of syntax. *ACM Computing Surveys*, 31, September 1999.
103. Dale Miller and Alwen Tiu. A proof theory for generic judgments: An extended abstract. In Phokion Kolaitis, editor, *18th Symp. on Logic in Computer Science*, pages 118–127. IEEE, June 2003.
104. Dale Miller and Alwen Tiu. A proof theory for generic judgments. *ACM Trans. on Computational Logic*, 6(4):749–783, October 2005.
105. Dale A. Miller, Eve Longini Cohen, and Peter B. Andrews. A look at TPS. In Donald W. Loveland, editor, *Sixth Conference on Automated Deduction*, volume 138 of *LNCS*, pages 50–69, New York, 1982. Springer.
106. Robin Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.
107. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part I. *Information and Computation*, 100(1):1–40, September 1992.
108. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, Part II. *Information and Computation*, 100(1):41–77, 1992.
109. Robin Milner, Joachim Parrow, and David Walker. Modal logics for mobile processes. *Theoretical Computer Science*, 114(1):149–171, 1993.
110. Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, 1991.
111. John C. Mitchell and Eugenio Moggi. Kripke-style models for typed lambda calculus. *Annals of Pure and Applied Logic*, 51(1-2):99–124, 1991.
112. Alberto Momigliano, Brigitte Pientka, and David Thibodeau. A case-study in programming coinductive proofs: Howe's method. Submitted, 2017.
113. J. Strother Moore. A mechanically verified language implementation. *J. of Automated Reasoning*, 5(4):461–492, 1989.
114. Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.
115. Gopalan Nadathur and Dustin J. Mitchell. System description: Teyjus — A compiler and abstract machine based implementation of λProlog. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 287–291, Trento, 1999. Springer.
116. Gopalan Nadathur and Debra Sue Wilson. A notation for lambda terms: A generalization of environments. *Theoretical Computer Science*, 198(1-2):49–98, 1998.
117. Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual model type theory. *ACM Trans. on Computational Logic*, 9(3):1–49, 2008.
118. Adam Naumowicz and Artur Korniłowicz. A brief overview of Mizar. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 67–72, 2009.
119. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer, 2002.
120. Bengt Nordstrom, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's type theory : an introduction*. International Series of Monographs on Computer Science. Oxford: Clarendon, 1990.
121. Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

122. Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer Verlag, 1994.
123. Lawrence C. Paulson. A generic tableau prover and its integration with isabelle. *J. UCS*, 5(3):73–87, 1999.
124. Alan J. Perlis. Epigrams on programming. *ACM SIGPLAN Notices*, pages 7–13, September 1982.
125. Frank Pfenning. Elf: A language for logic definition and verified metaprogramming. In *4th Symp. on Logic in Computer Science*, pages 313–321, Monterey, CA, June 1989.
126. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, June 1988.
127. Frank Pfenning and Ekkehard Rohwedder. Implementing the meta-theory of deductive systems. In *Proceedings of the 1992 Conference on Automated Deduction*, number 607 in LNCS, pages 537–551. Springer, June 1992.
128. Frank Pfenning and Carsten Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *16th Conf. on Automated Deduction (CADE)*, number 1632 in LNAI, pages 202–206, Trento, 1999. Springer.
129. Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In J. Giesl and R. Hähnle, editors, *Fifth International Joint Conference on Automated Reasoning*, number 6173 in LNCS, pages 15–21, 2010.
130. Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hricu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. *Programming Language Foundations*, volume 2 of *Software Foundations*. Online, 2010.
131. A. M. Pitts and M. J. Gabbay. A Metalanguage for Programming with Bound Names Modulo Renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction. 5th International Conference, MPC2000, Ponte de Lima, Portugal, July 2000. Proceedings*, volume 1837 of *LNCS*, pages 230–255. Springer, Heidelberg, 2000.
132. Andrew M. Pitts. Nominal logic, A first order theory of names and binding. *Information and Computation*, 186(2):165–193, 2003.
133. Andrew M. Pitts. Alpha-structural recursion and induction. *J. ACM*, 53(3):459–506, 2006.
134. Gordon D. Plotkin. A structural approach to operational semantics. *J. of Logic and Algebraic Programming*, 60-61:17–139, 2004.
135. The POPLmark Challenge webpage. `http://www.seas.upenn.edu/~plclub/poplmark/`, 2015.
136. François Pottier. Static name control for FreshML. In *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 356–365. IEEE, 2007.
137. Damien Pous. Weak bisimulation upto elaboration. In C. Baier and H. Hermanns, editors, *CONCUR*, volume 4137 of *LNCS*, pages 390–405. Springer, 2006.
138. Damien Pous. Complete lattices and upto techniques. In Zhong Shao, editor, *APLAS*, volume 4807 of *LNCS*, pages 351–366, Singapore, November 2007. Springer.
139. Damien Pous and Davide Sangiorgi. Enhancements of the bisimulation proof method. In Davide Sangiorgi and Jan Rutten, editors, *Advanced Topics in Bisimulation and Coinduction*, pages 233–289. Cambridge University Press, 2011.
140. Dag Prawitz. Hauptsatz for higher order logic. *Journal of Symbolic Logic*, 33:452–457, 1968.
141. Xiaochu Qi, Andrew Gacek, Steven Holte, Gopalan Nadathur, and Zach Snow. The Teyjus system – version 2, 2015. `http://teyjus.cs.umn.edu/`.
142. C. Röckl, D. Hirschkoff, and S. Berghofer. Higher-order abstract syntax with induction in Isabelle/HOL: Formalizing the pi-calculus and mechanizing the theory of contexts. In F. Honsell and M. Miculan, editors, *Proc. FOSSACS'01*, volume 2030 of *LNCS*, pages 364–378. Springer, 2001.
143. Davide Sangiorgi. π-calculus, internal mobility and agent-passing calculi. *Theoretical Computer Science*, 167(2):235–274, 1996.
144. Davide Sangiorgi and David Walker. *π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
145. Peter Schroeder-Heister. Rules of definitional reflection. In M. Vardi, editor, *8th Symp. on Logic in Computer Science*, pages 222–232. IEEE Computer Society Press, IEEE, June 1993.

146. Carsten Schürmann and Frank Pfenning. Automated theorem proving in a simple meta-logic for LF. In Claude Kirchner and Hélène Kirchner, editors, *15th Conf. on Automated Deduction (CADE)*, volume 1421 of *Lecture Notes in Computer Science*, pages 286–300. Springer, 1998.
147. Helmut Schwichtenberg. MINLOG reference manual. *LMU München, Mathematisches Institut, Theresienstraße*, 39, 2011.
148. Dana Scott. Outline of a mathematical theory of computation. In *Proceedings, Fourth Annual Princeton Conference on Information Sciences and Systems*, pages 169–176. Princeton University, 1970. Also, Programming Research Group Technical Monograph PRG–2, Oxford University.
149. Peter Selinger. The lambda calculus is algebraic. *Journal of Functional Programming*, 12(6):549–566, 2002.
150. Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(01):71–122, 2010.
151. Zachary Snow, David Baelde, and Gopalan Nadathur. A meta-programming approach to realizing dependently typed logic programming. In Temur Kutsia, Wolfgang Schreiner, and Maribel Fernández, editors, *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 187–198, 2010.
152. Mary Southern and Kaustuv Chaudhuri. A two-level logic approach to reasoning about typed specification languages. In Venkatesh Raman and S. P. Suresh, editors, *34th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 29 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 557–569, New Delhi, India, December 2014. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
153. Mary Southern and Gopalan Nadathur. A λProlog based animation of Twelf specifications. The International Colloquium on Implementation of Constraint and Logic Programming Systems (CICLOPS), 2014.
154. Aaron Stump. *Verified functional programming in Agda*. Morgan & Claypool, 2016.
155. Moto-o Takahashi. A proof of cut-elimination theorem in simple type theory. *Journal of the Mathematical Society of Japan*, 19:399–410, 1967.
156. Alwen Tiu. *A Logical Framework for Reasoning about Logical Specifications*. PhD thesis, Pennsylvania State University, May 2004.
157. Alwen Tiu. Model checking for π-calculus using proof search. In Martín Abadi and Luca de Alfaro, editors, *Proceedings of CONCUR'05*, volume 3653 of *LNCS*, pages 36–50. Springer, 2005.
158. Alwen Tiu. A logic for reasoning about generic judgments. In A. Momigliano and B. Pientka, editors, *Int. Workshop on Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP'06)*, volume 173 of *ENTCS*, pages 3–18, 2006.
159. Alwen Tiu and Dale Miller. Proof search specifications of bisimulation and modal logics for the π-calculus. *ACM Trans. on Computational Logic*, 11(2), 2010.
160. Alwen Tiu and Alberto Momigliano. Cut elimination for a logic with induction and co-induction. *Journal of Applied Logic*, 10(4):330–367, 2012.
161. Alwen Tiu, Gopalan Nadathur, and Dale Miller. Mixing finite success and finite failure in an automated prover. In *Empirically Successful Automated Reasoning in Higher-Order Logics (ESHOL'05)*, pages 79–98, December 2005.
162. Alwen Tiu, Nam Nguyen, and Ross Horne. SPEC: An equivalence checker for security protocols. In Atsushi Igarashi, editor, *Programming Languages and Systems: 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21 - 23, 2016, Proceedings*, pages 87–95. Springer International Publishing, 2016.
163. Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, 1990.
164. Christian Urban. Nominal reasoning techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008.
165. Christian Urban, James Cheney, and Stefan Berghofer. Mechanizing the metatheory of LF. *ACM Transactions on Computational Logic (TOCL)*, 12(2):15, 2011.
166. Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In R. Nieuwenhuis, editor, *20th Conf. on Automated Deduction (CADE)*, volume 3632 of *LNCS*, pages 38–53. Springer, 2005.
167. Jean van Heijenoort. *From Frege to Gödel: A Source Book in Mathematics, 1879-1931*. Source books in the history of the sciences series. Harvard Univ. Press, Cambridge, MA, 3rd printing, 1997 edition, 1967.

168. Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties*. PhD thesis, University of Pennsylvania, May 1996.
169. Björn Victor and Faron Moller. The mobility workbencha tool for the π-calculus. In *Computer Aided Verification*, pages 428–440. Springer, 1994.
170. Yuting Wang. *A Higher-Order Abstract Syntax Approach to the Verified Compilation of Functional Programs*. PhD thesis, University of Minnesota, December 2016.
171. Yuting Wang, Kaustuv Chaudhuri, Andrew Gacek, and Gopalan Nadathur. Reasoning about higher-order relational specifications. In Tom Schrijvers, editor, *Proceedings of the 15th International Symposium on Princples and Practice of Declarative Programming (PPDP)*, pages 157–168, Madrid, Spain, September 2013.
172. Yuting Wang and Gopalan Nadathur. A higher-order abstract syntax approach to verified transformations on functional programs. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 752–779. Springer, 2016.